

腾讯Android 自动化测试实战

汇集QQ浏览器、应用宝等亿级APP自动化测试精髓

丁如敏 盛娟◎等著



机械工业出版社
China Machine Press

腾讯Android自动化测试实战

丁如敏 等著

ISBN: 978-7-111-54875-1

本书纸版由机械工业出版社于2016年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

本书编委

序

前言

第1章 概述

1.1 Android自动化测试框架概述

1.2 本书内容概述

第2章 自动化测试框架及应用领域综述

2.1 自动化测试框架介绍

2.1.1 一个简单的Android App自动化测试过程

2.1.2 自动化测试框架基本原理

2.2 移动终端自动化测试应用场景

2.3 本章小结

第3章 Robotium框架工作原理及实践

3.1 Robotium常用功能

3.1.1 什么是Robotium

3.1.2 Robotium提供的类

3.1.3 环境搭建

3.1.4 Robotium的控件获取、操作及断言

3.2 Robotium原理简析

- 3.2.1 Robotium支持Native原理
- 3.2.2 Robotium支持WebView原理
- 3.3 Robotium实践运用
 - 3.3.1 控件ID相同时获取控件
 - 3.3.2 ListView列表遍历
 - 3.3.3 修改Robotium以支持X5WebView
- 3.4 本章小结

第4章 Monkey基本原理及扩展应用

- 4.1 Monkey基础知识
 - 4.1.1 Monkey概况
 - 4.1.2 Monkey参数
 - 4.1.3 Monkey事件
 - 4.1.4 Monkey环境搭建
 - 4.1.5 Monkey启动
- 4.2 Monkey测试方法
 - 4.2.1 Monkey测试实例
 - 4.2.2 Monkey日志分析
- 4.3 Monkey的基本原理
 - 4.3.1 Monkey代码框架
 - 4.3.2 Monkey代码逻辑详解
- 4.4 Monkey扩展应用示例

- 4.4.1 Monkey代码重编译执行方法
- 4.4.2 Monkey截图优化
- 4.4.3 Monkey Wi-Fi自动重连优化
- 4.4.4 Monkey扩展应用的优点和缺点

4.5 本章小结

第5章 UIAutomator框架及实践

5.1 UIAutomator简介

5.2 UIAutomator解读

5.2.1 UIAutomator框架解读

5.2.2 UIAutomator原理解读

5.2.3 UIAutomator API解读

5.3 UIAutomator实战

5.3.1 UIAutomator快速上手

5.3.2 UIAutomator设计思想

5.3.3 UIAutomator实践案例

5.4 UIAutomator总结

5.4.1 UIAutomator代码规范及建议

5.4.2 UIAutomator技巧及封装

5.5 本章小结

第6章 Appium框架解析及实践

6.1 Appium框架概况

- 6.1.1 Appium架构原理
- 6.1.2 Appium框架的优缺点
- 6.2 Appium框架工作解析
 - 6.2.1 Appium环境搭建
 - 6.2.2 HelloWorld测试示例
 - 6.2.3 Desired Capabilities的说明
 - 6.2.4 Appium API的解读
- 6.3 Appium框架在腾讯地图中的实践
 - 6.3.1 Appium接口的封装
 - 6.3.2 测试脚本设计思想
 - 6.3.3 Appium在腾讯地图中的测试实践
 - 6.3.4 Hybrid App的测试方法
 - 6.3.5 Appium脚本常见问题及处理方法
- 6.4 本章小结

第7章 Android App速度测试

- 7.1 速度测试场景
- 7.2 速度测试的六大方法
 - 7.2.1 掐表计时法
 - 7.2.2 打印日志计时法
 - 7.2.3 图像分析计时法
 - 7.2.4 Hook方案计时法

- 7.2.5 网络包分析法
- 7.2.6 各种速度测试方法的优缺点
- 7.3 手机QQ浏览器网页打开速度测试实践案例
 - 7.3.1 确定关键指标
 - 7.3.2 选择测试方法
 - 7.3.3 整体方案
 - 7.3.4 解决关键问题
 - 7.3.5 速度优化效果
- 7.4 手机QQ浏览器多窗口按钮速度实践案例
 - 7.4.1 为什么要做多窗口按钮速度测试
 - 7.4.2 什么是多窗口按钮速度测试
 - 7.4.3 多窗口按钮速度测试影响因素和测试方法
 - 7.4.4 如何进行多窗口按钮速度测试
- 7.5 本章小结

第8章 视频性能测试案例

- 8.1 视频性能测试需求分析
- 8.2 视频首帧性能测试方案的设计思路
 - 8.2.1 视频播放流程
 - 8.2.2 设计思路
- 8.3 视频首帧性能测试方案的具体实现
 - 8.3.1 开发工具准备

- 8.3.2 测试环境准备
- 8.3.3 工程部署
- 8.3.4 关键代码和难点分析
- 8.3.5 编译环境配置
- 8.3.6 工具安装

8.4 方案优缺点

8.5 本章小结

第9章 应用宝BVT测试案例

9.1 测试工程

- 9.1.1 测试工程概览
- 9.1.2 测试工程签名

9.2 测试用例

- 9.2.1 测试用例生命周期
- 9.2.2 测试用例编写
- 9.2.3 测试用例执行
- 9.2.4 测试用例管理

9.3 测试报告

- 9.3.1 Spoon介绍
- 9.3.2 结合Spoon的出错重试与截图
- 9.3.3 结合Spoon生成汇总报告

9.4 Robotium跨应用

9.4.1 UIAutomator Dump方式跨应用

9.4.2 UIAutomator结合Instrumentation模式

9.5 代码覆盖率

9.5.1 覆盖率定义

9.5.2 覆盖率工具

9.5.3 JaCoCo介绍与实践

9.5.4 BVT测试与覆盖率结合

9.5.5 指导建议

9.6 本章小结

第10章 兼容性测试实践

10.1 兼容性测试概述

10.2 兼容性测试方法

10.2.1 手动测试

10.2.2 自动化测试

10.2.3 云平台测试

10.3 兼容性测试思考

10.4 本章小结

本书编委



丁如敏

毕业于北京邮电大学，近10年的软件测试和项目管理经验，精通移动终端性能测试、自动化测试、敏捷测试等各种测试技术。在腾讯工作期间，带领团队共发明国家级专利50多项，开发10多门内部培训课程。喜欢挑战软件领域的各项前瞻技术，并有丰富的实践经验。



盛娟

毕业于合肥工业大学计算机及应用专业，腾讯科技高级测试工程师。之前先后服务于中国联通、CISCO中国研发中心，有10多年的软

件测试和项目管理经验。近两年主要负责搭建QQ浏览器Android端质量保证体系，积累了丰富的移动终端项目经验。



陈航特

毕业于南京理工大学电子信息工程专业，专注于Android端自动化测试，是国内较早进入该领域的探索者，有丰富的使用与二次开发Android原生自动化测试框架的实战经验。目前负责腾讯应用宝的客户端自动化测试与相应的平台搭建工作。



陈六四

腾讯高级工程师，12年软件开发和测试工作经验，曾任职于多家知名跨国企业，现任职于腾讯公司，从事浏览器视频相关测试开发工

作，并在视频领域取得多项国家级专利。擅长工具开发、前端性能测试和后台性能测试。



邓曦

毕业于电子科技大学，现任职于腾讯公司，担任无线研发部工程师，负责手机QQ浏览器（Android）测试工作。在测试领域拥有超过8年的经验，在自动化测试方面经验丰富。帮助手机QQ浏览器（Android）实现了从零到行业第一的飞跃。



高苡新

2006年北航本科毕业，10年软件测试经验、5年移动互联网测试经验，长期负责手机QQ浏览器性能测试，2013年至2015年专职负责网页

打开速度测试。从零开始搭建手机QQ浏览器速度自动化测试方案。其所在的自动化团队获得2013年MIG移动互联网事业群SEVP特别奖。



林凯杰

腾讯专项技术测试工程师，主要从事ROM级别的App测试工作，在Android自动化方面有3年实践经验，对跨应用自动化实现有许多心得。自动化实现方式有很多，因地制宜最重要！



刘洋

腾讯高级测试工程师，毕业于大连交通大学。加入腾讯前分别在华为、中国移动担任过系统测试工程师，主要从事网络、移动领域的工作。于2009年加入腾讯移动测试组，主要致力于精准、代码耦合、

覆盖率、自动化等方面的研究与实施，擅长Linux、Android相关工作。



鲁万林

毕业于武汉大学，曾任职于华为公司，现任职于腾讯公司，担任无线研发部高级工程师。手机QQ浏览器Android平台测试负责人，在测试领域有10多年的经验，在成都从无到有建立了一只强大的浏览器测试团队，帮助QQ浏览器从零飞跃到行业第一。



万宇

2006年毕业于浙江大学计算机系。10年一流公司工作经验，先后在SAP、Oracle、腾讯等公司从事测试开发工作，目前负责Android手

机QQ浏览器等产品的自动化测试开发工作。擅长工具开发，对性能测试、自动化测试有深入的理解。



郑若琳

2009年毕业于广东财经大学。加入腾讯4年，负责过QQ浏览器国际版、新蜂ROM、手机应用宝等移动端产品的业务测试和自动化测试，目前负责应用宝的测试体系建设和测试外包管理工作。具有多年测试实战经验，擅长自动化测试工具运用。



钟书成

毕业于成都信息工程大学和中国科学院，腾讯高级测试工程师。加入腾讯前曾在多个外企项目中从事测试开发工作，于2012年加入腾

讯地图项目，主要致力于自动化测试的研究与实施，在Android自动化测试方面有丰富的经验。在进行腾讯地图项目期间还负责八爪鱼自动化测试平台的设计与开发工作。

序

最近和腾讯移动品质中心（TMQ）接触比较多，除了技术的交流，还邀请TMQ资深人士参加了某个软件工程论坛并做了分享，关注了TMQ公众号。现在很高兴为这个优秀团队的新书《腾讯Android自动化测试实战》写序，因为可以先睹为快，提前学习腾讯的经验。

现在移动应用很普及了，无须摆事实、讲道理，读者都深有体会。但10年前，移动应用还相对落后，那时TMQ就已经开始专注移动App的测试，故这个团队在移动应用专项测试、精准测试体系及自动化测试方面都有着丰富的实战经验。这本书就是他们2015年策划的移动测试领域的3本新书之一。这本书专注Android自动化测试，覆盖了从环境配置、UI元素获取、用例编写到脚本开发、编译、执行等整个移动应用的生命周期。针对常用的Android自动化测试框架和工具，如Appium、Monkey、Robotium和UIAutomator等都进行了详细介绍，从其原理简析开始，循序渐进地介绍了其安装、设置以及API调用等知识，并围绕着实例详细介绍了其应用实践、技巧，读者一面看书、一面实践，就能轻松掌握Android自动化测试的技能。

虽然是小小的App应用，涉及的技术却不比桌面或Web低，反而由于资源更宝贵、网络连接不稳定、迭代更快、用户体验要求更高等，在单元测试、性能测试、压力测试、兼容性测试、速度测试等各方面

都更具挑战性，测试人员还要面对Native、WebView和HTML5等不同技术。本书对上述所有内容，包括一些具体的技术细节，如非耦合式用例设计、API接口的封装等，都有很好的交代。书中还提供了完整的实例，从测试工程概览、签名开始，到测试用例编写、执行、管理，再到结合Spoon生成汇总报告，一气呵成。

注重品质的团队，写起书来也绝不会忽视质量，这本书就是一个典范。TMQ将书的质量放在首位，不仅选择最有经验的测试工程师组成一支很强的写作团队，而且初稿出来之后经过了6轮的内部评审，参加评审的人员之多、评审时间之长，是绝无仅有的，因此这样写出来的书，质量是有保证的。

本书不仅介绍了Android自动化框架的基础知识、原理和API使用，而且分析过程逻辑清楚，设计和实现思路清新自然，还触及一些较深的主题，如框架的二次开发等，故本书适合不同层次的测试人员和开发人员学习。借助网站的在线支持，本书如虎添翼，更加保证了读者的学习效果。

综上所述，本书是一本值得向大家推荐的好书，大家一定会喜欢的。有了“她”，轻松完成Android自动化测试也就不在话下了。

朱少民

于上海

前言

为什么要写这本书

早在2010年年底，我们团队就有出一本关于移动互联网测试书籍的计划（那时候移动互联网测试书籍基本没有），当时计划的内容涉及面比较广，涵盖测试设计、测试用例管理、测试流程、自动化测试、专项测试等领域。不过，由于各种原因被搁浅，确实有点儿可惜，否则移动互联网测试国内的第一本书当时就面世了。这次终于又有机会整理这些年的测试经验并形成一本书了，借此可以跟业界的同行一起交流切磋。

TMQ（Tencent Mobile Quality）腾讯移动品质中心，是腾讯内部最早专注于移动App测试的团队，在10余年的时间内承担了近10款业界领先产品的测试工作，为腾讯向移动方向转型提供了多项质量方案和关键专利。本书的作者都是TMQ平台的核心成员，服务于公司级的手机QQ浏览器、应用宝等项目，经过这几年来在移动测试领域的探索与实践，摸索出了一些实实在在的实践经验。TMQ的老板鼓励我们把这些知识和经验编写成册，这样不仅能为公司内部提供好的产品或服务，也能为业界同人提供参考，从而将知识更好地扩散出去。我们团队非常珍惜这次写书的机会，故组织了团队内Android自动化测试方面经验

丰富的同学一起来编写。大家都是利用自己的业余时间总结各自擅长领域的经验和知识，且初稿经过6轮的内部评审（参加评审的同事超过30位），前后历时半年多才最终完成了本书的写作和修改，希望能给读者提供一本质量较高的专业书籍。

TMQ经过这几年的积累，在专项测试、精准测试体系及自动化测试方面都有比较多的精辟之作。基于此，我们分小组对这些知识进行了整理，形成了相应的知识库，完成了本系列丛书，包括《移动互联网App性能评测调优实践》《精准化测试白皮书》《腾讯Android自动化测试实战》3本书，其他两本也在同期编写出版工作中。希望TMQ平台出品的书籍能给予读者思路上的指导或者是技术上的解惑。

除封面署名外，参加本书编写的作者还有：陈航特、陈六四、邓曦、高苡新、林凯杰、刘洋、鲁万林、万宇、郑若琳、钟书成共12位作者（按姓氏拼音排序），都是来自腾讯移动端QQ浏览器及应用宝团队的骨干员工。

读者对象

本书是一本务实的书籍，案例都是作者们的第一手资料，对于软件质量保证方面的初学者，本书还提供了简短的案例以帮助其理解，循序渐进，掌握测试核心原理；对于有经验的同行，本书提供了经典

案例帮助其提升与参考。这里根据行业实际需求给出了相应的用户群体：

- 对移动业务测试感兴趣的人；
- 对Android自动化测试感兴趣的人；
- 即将开展Android自动化测试的团队；
- 开设相关课程的院校师生。

本书特色

Android自动化测试经过这几年的发展，官方提供的开源自动化框架已经能非常好地支持终端的测试业务，但是如何利用、如何用好这些资源还是比较现实客观的问题，尤其是在小公司中，自动化测试方法的摸索及实施还存在一些困难，需要一定的投入才能得以真正应用。业界一些Android自动化测试方面的书籍很多都偏原理的介绍，而本书不仅深度解析这些框架的原理，还给出了手机QQ浏览器、应用宝项目中的典型案例，像最常见的App速度、要求较高的视频播放性能测试等，供需要实践的读者学习，这也是本书的重要特色之一。本书前半部分主要介绍业界流行的Android自动化框架的基础知识，聚焦工具框架的原理以及基础API使用、框架的二次开发改造（根据具体项

目做相应修改)，以及实践过程中一些共性问题的分享。如果读者已经掌握这些框架基础，那么对本书内容的理解就会更容易。同时读者可以重点关注本书中介绍的对框架进行二次开发的内容，并结合自己的实际项目考虑如何应用这些知识提升自己的工作效率；基础比较高的读者可跳过这部分直接阅读后半部分。后半部分通过一些实际案例来讲解自动化框架的应用，更强调系统性分析设计能力，包括需求的分析、工具选型、测试方案、代码覆盖率的应用等，覆盖功能测试、性能测试的具体实战案例。这部分对读者的技术能力要求相对更高一些，涉及的知识点的深度和广度要明显高于前半部分，需要进行

Android App应用的性能速度测试的读者可以深入阅读，领会书中所提场景的测试设计与思路，进而掌握框架的精髓所在。在经典案例中也给出了很多具体实现思路的介绍与分析，让读者知其然、并知其所以然，同时各位作者也把项目测试工程代码加以整理，打包至**TMQ**后台，供读者下载，读者如有需要可以直接导入工程进行调试学习，以大大减少学习成本。读者可以根据自己的需求阅读相应章节的内容：如熟悉Java语言，又面临Debug未混淆被测App的情况，建议直接学习**Robotium**框架，因为**Robotium**操作简单、相关资料丰富，还能支持ant、maven打包，与jenkins结合较好；因**Robotium**不支持跨应用，所以对于需要支持跨应用的框架，读者可以阅读**UIAutomator**和**Appium**框架，其中**Appium**是借助WebDriver JSON协议实现的，能支持多种语言编写测试脚本；对于有一定经验的读者，在案例选择时可以结合

Robotium和UIAutomator的优点一起使用，此时可直接阅读本书中的浏览器视频性能测试案例。

勘误和支持

本书作者分别在深圳、北京、成都、合肥四地办公，所以整个写作过程中异地沟通比较多，整书从选题至初稿形成，差不多花了5个月的时间，速度超过我们的预期，但由于作者的水平有限、异地交流、编写时间仓促等，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。为此，我们特意在TMQ的官网创建了一个在线支持：<http://tmq.qq.com>。你可以将书中的错误发布在新书勘误表页面中，我们将尽量在线上为读者提供最满意的解答。书中的全部源文件除可以从华章官方网站（<http://www.hzbook.com>）下载外，还可以从TMQ网站下载，我们也会将相应的错误及时更正。如果你有更多的宝贵意见，也欢迎发送邮件至邮箱dinahsheng@tencent.com，期待能够得到你们的真挚反馈。

致谢

感谢腾讯科技MIG无发研发部总经理冼文佟、助理总经理陈诚，正是他们鼓励我们多总结、多分享、把知识传播，我们才有了写书的想法。

感谢腾讯科技MIG无发研发部品质中心的总监廖志、李德广、张鼎，给我们提供TMQ这样好的一个人才培养平台，让我们不断成长和提升，同时也非常感谢他们在百忙之中指导我们写作。

感谢腾讯科技的同事们在整个写作过程中，帮我们多次进行内容及技术层面的审核指错，尤其是王琳同学，作为每章内容的第一位读者，给了很多好的建议和思路，其他同事也对本书的内容进行了细致的评审，他们是陆小三、吴景、柳炜、沈东雄等，在此一并谢过。

感谢腾讯科技的浏览器测试团队、应用宝测试团队，在我们编写过程中他们提供的工作支持，是我们得以这么快速完成的基础。

感谢同济大学朱少民教授给我们的专业指导和鼓励，让我们更有信心完成本书的编写。

感谢机械工业出版社华章公司的编辑杨福川、孙海亮，是他们多次给予我们指导及鼓励。

感谢我们的亲人在我们周末加班写作时给予的支持与理解，你们的支持是我们最大的动力！

谨以此书献给我们最亲爱的家人，以及热爱移动业务测试的朋友们！

丁如敏等

第1章 概述

在展开各章节简介之前，本章先带读者了解一下Android自动化测试框架的大体历史以及框架的演进过程。Android自动化测试框架和工具从2009年发展至今日趋成熟，从早期官方提供的半自动化演进到全自动化框架，包括支持跨应用、WebView等，其功能越来越强大，并融合库思想、数据驱动、模块化、函数桩等先进的自动化测试思想和理念，Android测试起来越便捷。本章主要介绍Android App自动化框架的历史及热点问题。

1.1 Android自动化测试框架概述

2007年Android开源时，Monkey、Instrumentation和MonkeyRunner这3个测试框架，是跟Android源码一起发布的，这也是最早可用的自动化测试框架，那几年大家基本都是用这些框架来开展自动化相关测试工作的。2010年，第一个第三方的测试工具Robotium（基于Instrumentation）发布了，不少测试人员就转用这个框架，Robotium社区逐步发展起来。图1-1所示为Robotium热度随时间变化的趋势。

2010年还有一个自动化测试框架Robolectric开源了，主要支持单元测试；Robolectric允许用户做大部分真实设备上可以做的事情，且可以在常规的JVM持续集成环境中运行，不需要通过模拟器，因此可以摆脱模拟器启动慢的问题。

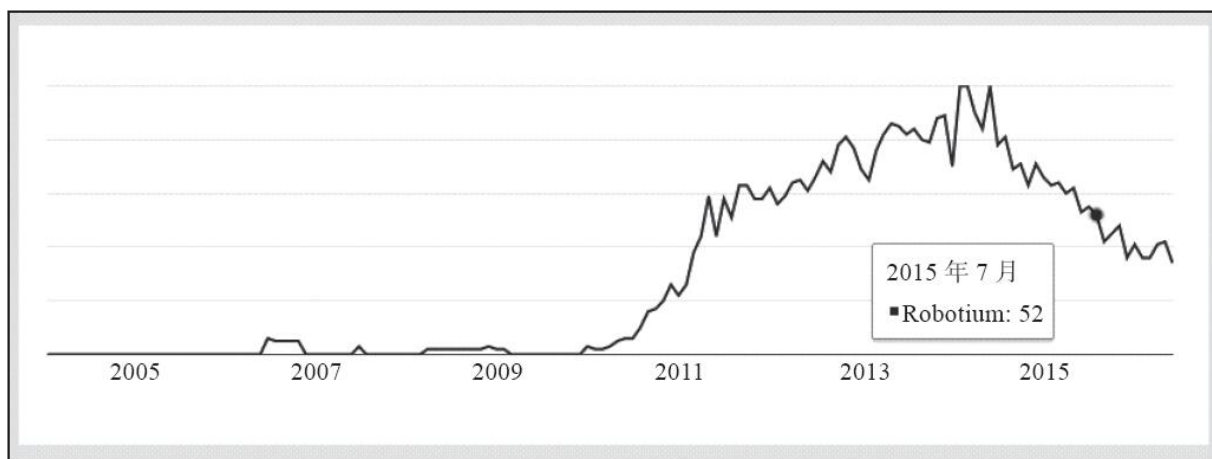


图1-1 Robotium热度随时间变化的趋势

注：引用自<https://www.google.com/trends/explore#q=robotium>

2011年新发布的Android SDK包含了chimpchat，可以通过Java来调用Monkey，也可以用Java语言实现类似MonkeyRunner功能（MonkeyRunner之前只支持Python）。

2013年则是一个Android自动化测试框架爆发年，Selendroid、Espresso、Calabash、Appium等框架都是在这一年发布或者开源的。其中，Appium整合Selendroid以及UI Automator等框架的优点，使用Selenium的WebDriver JSON协议，使WebDriver用户使用起来非常方便。自2013年以来，Appium发展非常迅速（不过基本上是印度的同学在搜索）。图1-2所示为Appium热度随时间变化的趋势。

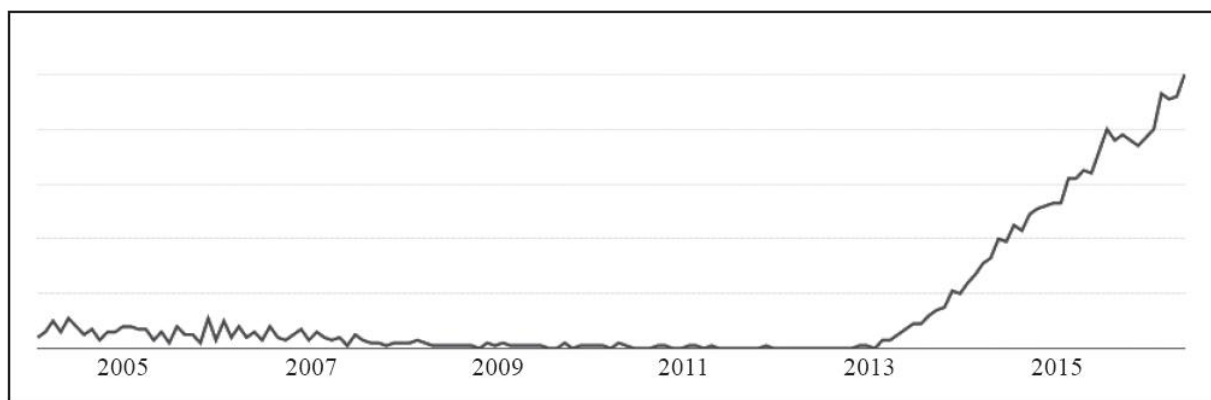


图1-2 Appium热度随时间变化的趋势

注：引用自<https://www.google.com/trends/explore#q=Appium>

按照时间线，把上面介绍的自动化测试框架梳理一下，如图1-3所示。

当然国内也有不少团队或者个人开发自动化测试框架并开源，例如百度的Café（<https://github.com/BaiduQA/Cafe>），阿里巴巴的两个自动化测试框架Athrun（<http://code.taobao.org/svn/athrun>）、Macaca（<http://macacajs.github.io/macaca/>）。腾讯内部也开发了Android自动化框架，不过暂时没有开源。相信国内其他公司也在开发相关自动化框架，有些公司基于已有的开源框架二次开发定制适合自己项目的自动化框架，可能暂时也没有开源。

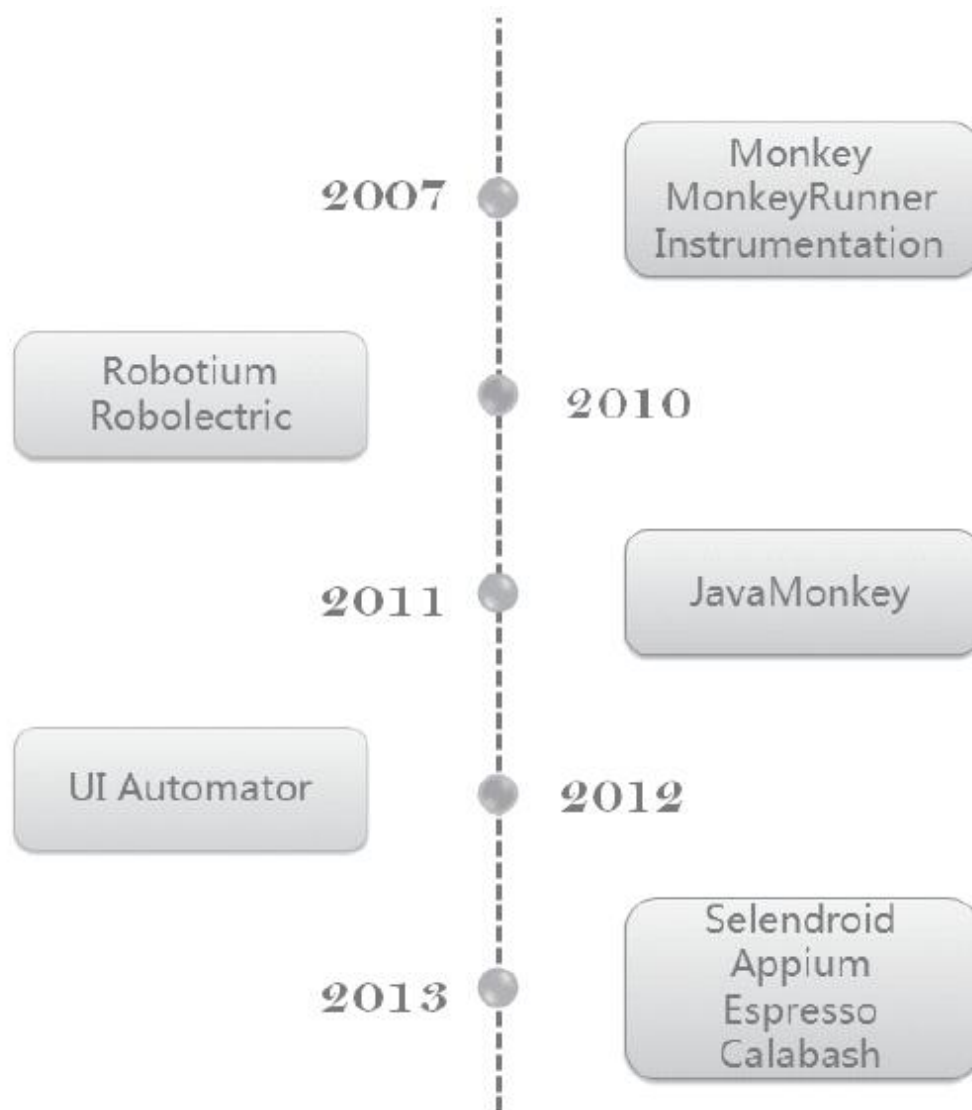


图1-3 Android自动化测试框架时间线

注：上面提到的是相对有一定使用人数的自动化测试框架，除此之外，业界还有不少其他测试框架，如MonkeyTalk、RoboSpock、NativeDriver等。

上面简单介绍了Android自动化测试框架的历史（时间线展示）。Android自动化测试里面还涉及到一个签名的问题（Instrumentation的限

制），我们按照重签名的维度重新划分一下，方便读者做自动化测试框架的选型。图1-4所示为Android自动化测试框架图谱，仅供参考。

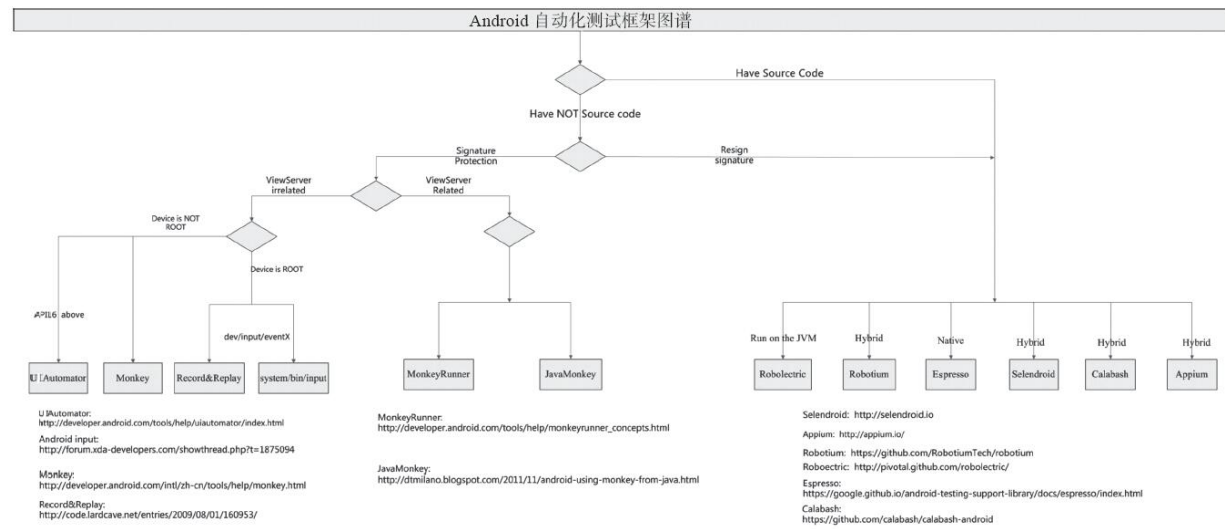


图1-4 Android自动化测试框架图谱

注：Appium不需要更改被测App签名；

Hybrid：混合Android原生控件以及Webview控件；

Native：纯Android原生控件。

1.2 本书内容概述

本书主要介绍Android自动化测试相关内容，分为以下两大部分。

第一部分介绍业界流行的Android自动化框架的基础知识，聚焦工具框架的原理和基础API使用，以及框架的二次开发（根据具体项目做相应修改），分享实践过程中的一些共性问题。这部分内容需要读者有基本编程能力（主要是Java方面的编程基础），如果读者已经具有这方面的能力（比如独立按Android开发者官网文档搭建相关HelloWorld的工程），那就更容易理解书本的内容；同时可以重点关注如何进行二次开发，以及如何应用于实际项目。

第二部分通过一些实际案例来讲解这些自动化框架的应用，更强调系统性分析设计，包括需求的分析、工具选型、代码覆盖率的应用，以及覆盖功能测试以及性能测试的具体实战等。这部分对读者的技术能力要求相对更高一些，涉及的知识点会多一些。当然如果有不太明白的地方，欢迎与各章节的作者进行交流。

第一部分Android自动化测试基础篇。

本书第一部分内容就是从自动化框架图谱中选择业界相对流行的自动化测试框架展开介绍，根据Google上这些自动化测试框架搜索量以及我们了解到的各互联网公司的自动化框架使用情况，我们选择有

代表性的4个框架（Monkey、Robotium、UI Automator、Appium）分别进行介绍。第1章也即本章，所以从第2章开始简介。

第2章自动化测试框架及应用领域综述：本章通过一个浅显易懂的Android自动化测试案例进行详细讲解，提炼出通用的自动化测试框架的原型——“动作执行/结果判断/报告展示”，最后介绍自动化测试的各种应用场景，方便读者更好地应用自动化测试。

第3章Robotium框架工作原理及实践：本章要求读者有一定的Robotium基础，可以先到Robotium官网下载相关Robotium的基础Demo练习。第一节先简单介绍Robotium的特点以及优缺点，再到Robotium主要API的详解，对Native控件以及Webview控件分别从获取以及相关操作展开介绍，同时还有不少实践过程中技巧的分享，例如搜索以及等待时间、截图、断言等。第二节深入Robotium的代码框架，结合代码分析，分别介绍控件获取以及操作的具体实现原理；针对最近又流行起来的H5页面，着重介绍Webview的基本原理，方便读者对H5页面进行测试。第三节通过分享3个实际案例（都是测试人员会经常碰到的案例），方便读者处理类似问题。第一个案例是解决相同ID或者没有ID的控件；第二个案例是对ListView在屏幕之外操作技巧；第三个案例则是针对一些定制化的Webview（例如腾讯浏览器服务X5Webview）进行适配，让Robotium也能快速支持定制化Webview的自动化测试。

第4章Monkey基本原理及扩展应用：每个从事过Android App测试的同学都应该使用过Monkey工具。Monkey工具也是最基础的自动化工具之一，上手比较容易，因此基本是大家的首选。本章第一节从Monkey基础知识开始介绍，从参数配置到环境搭建做基本解读，让读者能够通过本小节启动自己的Monkey自动化测试。第二节通过测试实例讲解Monkey具体使用，以及使用Monkey发现crash后产生日志的统计分析，解决实际测试过程中的一些问题，让读者能结合自己的项目做Monkey自动化测试。第三节通过Monkey的源码来介绍Monkey的代码基本框架以及某些逻辑详解，让读者清楚地了解Monkey运行逻辑（需要用户有相关Java代码基础），使得读者“知其然更知其所以然”。Monkey提供的功能可能没法满足要求，我们需要通过对Monkey的二次开发来定制需求，第四节通过两个实际案例来详细介绍Monkey的二次开发过程，这样就方便读者动手二次开发自己的需求。

第5章UI Automator框架及实践：本章第一节先简单介绍UI Automator的发展历程以及特点，让读者有一个基本认识。第二节介绍UI Automator整体框架、UI Automator各个类以及它们之间的关系，然后重点介绍五大基础类UiDevice、UiSelector、UiObject、UiCollection、UiScrollable。接下来重点介绍该框架两个重要事情——界面解析和事件注入，通过代码解读这两块的基本原理。最后把API都注解一下，方便读者查询。第三节主要通过实战案例来展示UI Automator的使用，从功能测试到性能测试以及压力测试都有相关案例

讲解，基本测试类型自动化都可以选择UI Automator来完成；同时总结使用过程中的问题，如输入法、第三方包编译、adb稳定性等问题，方便读者借鉴思路。

第6章Appium框架解析及实践：本章第一节介绍Appium基本架构原理以及使用到的一些技术点，同时说明Appium框架的优缺点，让读者有一个大概认识，方便做自动化测试框架选择。第二节从环境搭建入手，手把手教读者完成一个HelloWorld的测试示例，接下来针对日常可能用到的方法对API进行解读，让读者逐步上手。第三节开始介绍自动化测试一些进阶思路，例如接口封装以及用例设计思想，引导读者把自动化测试做得更好。接下来以腾讯地图自动化测试实践为案例，分别从可重复性、稳定性和可维护性三个方面详细介绍，同时对Hybrid App测试做了介绍，最后对Appium实践过程中常见问题做了解答，方便读者借鉴，避免走弯路。

第二部分Android自动化测试实战篇

第7章Android App速度测试：本章选择对用户体验感知明显的一个性能点——App速度，针对App速度测试来进行深度分析，从整个需求的系统分析，再到详细对比不同速度测试方法（掐秒表、打Log、录像、Hook方式、网络包分析等），最后提炼出速度测试的相关方法论，以供读者参考。

第8章视频性能测试案例：本章选择视频性能这个需求进行系统性分析，从用户感知层面出发，挖掘相关需求点，再到整体性能方案的设计（从自动化执行以及结果对比等都做详尽的分析），最后到结果分析等方面，都做了系统性详细介绍，给读者一个完整的案例。这一章对读者基础能力要求有点儿高，除了涉及Java编程语言，还涉及C的编程（JNI），同时要求在视频方面有一定的基础（如FFMPEG）。

第9章应用宝BVT测试方案：Robotium在应用宝项目的实践案例。主要介绍如何利用Robotium来做BVT（Build Verification Test），把App最基础功能梳理出来，然后写成自动化测试用例，每次持续集成编译成功，都会运行这些自动化脚本，确保每个版本基础功能可用。同时结合代码覆盖率，可以关注覆盖度情况。代码覆盖率主要用JaCoCo生成，当然Emma也使用很广泛。

第10章兼容性测试实践：Android手机从2007年发布，到目前为止，有超过2000种型号，系统从2.X到6.X，Android碎片化比较严重，那么怎么保证App在大多数的机型系统中正常运行呢？这一章介绍如何利用业界的云测试系统进行测试。云测试系统包括百度MTC、Testin、优测，通过使用这些云测试系统可以快速发现一些兼容性问题。

第2章 自动化测试框架及应用领域综述

近几年，随着移动互联网的快速发展，智能终端的App应用越来越广，Android测试技术也备受重视，新的终端自动化测试框架层出不穷，本章笔者就自动化测试的入门知识及其应用做一个浅显的梳理与总结，与读者一同探讨移动终端自动化测试思路 and 方案。同时，本书主要也是围绕本章节提到的基础框架及其应用场景进行实战分析与演练，以亲身体会总结出实际项目经验，给准备实施或正在实施自动化测试的读者提供一些帮助和建议。

自动化测试在软件测试的各大沙龙、行业峰会以及培训课程中都是一个热门的话题，很多测试人员也非常热衷于自动化前瞻知识的研究和学习。那么Android自动化测试框架如何理解和定义呢？每个人给出的答案不一定完全一样，笔者认为，狭义上就是通常说起的自动化测试框架，像很早就风靡的Robotium、后起之秀UI Automator、跨平台的Appium以及类似于Monkey的稳定性工具等，广义上包括自动化测试框架和测试管理平台；前者通用于狭义概念上的理解，后者主要是测试中对测试用例、执行、资源调度、问题提单、数据统一存储、报告输出等进行综合的展示平台。本书主要介绍基于狭义定义的自动化实践，本章知识结构图如图2-1所示。



图2-1 本章知识结构图

2.1 自动化测试框架介绍

2.1.1 一个简单的Android App自动化测试过程

在了解相关的Android App的自动化测试框架之前，先来看一个常用的自动化测试实例，这里先不讨论框架，主要是测试用户操作的模拟、执行结果的判断，以便获得对测试自动化的感性认识。

案例需求如下：QQ浏览器打开手机存储卡的文件，通过自动化测试获取其打开某一文件的响应时间，这里首先需要做细分，把需求拆分为几个关键点，即进入浏览器、文件打开操作、获取手机屏幕、截图分析、结果统计输出。自动化测试就是实现机器完成这些关键点的一系列操作，并且在脚本的实际运行中添加需要的业务逻辑判断，实现测试自动化。根据脚本的具体实现，整理出打开文件测试流程图，如图2-2所示。

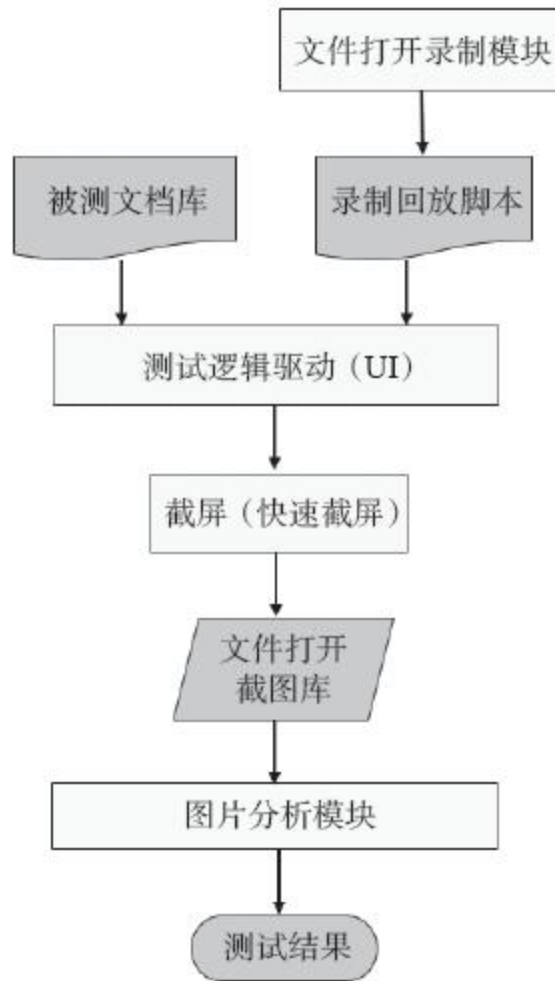


图2-2 打开文件测试流程图

1.准备测试工具

下载record和replay的脚本录制工具，有很多工具可以实现，比如本书中的MonkeyRunner、UIAutomator等，在这个案例里从网上下载record/replay可执行工具，直接复制至手机硬盘。

2.录制测试脚本

在PC上连接手机，打开adb shell命令，进入record工具存放目录，用shell命令运行record工具启动用户脚本录制，并给录制命名，如wps（打开wps文件）。工具运行成功之后，手工完成所需要的业务操作（如手机主页→选择QQ浏览器→文件目录→双击wps文件），操作结束后，按Ctrl+C键结束脚本的录制工作。录制脚本过程如图2-3所示。



```
D:\>adb shell /data/local/tmp/record enter wps
check and create directory:0...
```

图2-3 录制脚本过程

3.执行测试脚本

这里以Java语言进行测试脚本的编写作为示范，通过Java的ProcessBuilder类在PC上创造并启动一个cmd命令行的进程，并执行“adb shell replay”操作进行脚本回放，这样打开文件的功能就可以通过脚本完成相应的操作执行，如代码清单2-1所示。

代码清单2-1 实现浏览器中打开wps文件

```
public static synchronized void enterWPS()
{
    ProcessBuilder pb = new
    ProcessBuilder("cmd.exe");//ProcessBuilder("/system/bin/sh");
    // java.lang.ProcessBuilder: Creates operating system processes.
    pb.directory(new File("C:\\"));
    // 设置

    shell的当前目录。

    try {
        Process proc = pb.start();
```

```
BufferedReader in = new BufferedReader(new
InputStreamReader(proc.getInputStream()));
// 获取输入流, 可以通过它获取
```

SHELL的输出。

```
BufferedReader err = new BufferedReader(new
InputStreamReader(proc.getErrorStream()));
PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter
(proc.getOutputStream())), true);
// 获取输出流, 可以通过它向
```

SHELL发送命令。

```
out.println("adb shell replay enter wps");
//打开
```

WPS文件

```
out.println("exit");
String line;
while ((line = in.readLine()) != null)
{
    System.out.println(line);
}
while ((line = err.readLine()) != null)
{
    System.out.println(line);
}
in.close();
out.close();
proc.destroy();
}
catch (Exception e)
{
    System.out.println("exception:" + e);
}
}
}
```

这里的录制脚本存放路径需要读者在实践时进行更改, 或者把录制的脚本放置在/system/bin/。

4.结果判断

这里通过测试脚本完成了用户操作的模拟实现，但是正常的测试还需要结果的验证，需要编码脚本进行测试结果的判断，本案例可以通过截屏和图片分析wps文件是否打开成功，可以通过Google汉明距离相似度对比算法得到图片相似度来判定打开的结果，代码相对要复杂一些，后面的案例中也有类似的代码实现，这里就不再细述。

2.1.2 自动化测试框架基本原理

经过前面的一个简单的自动化测试案例，我们对**Android**的自动化测试有了一个感性的认识，很多有相关工作经验的测试同学也都会理解，这和**PC**的自动化测试思路是相通的，只不过所借助的框架不同，目前业界已经有很多成熟的开源**Android**端自动化测试框架，经常用到的框架代表有**Robotium**和**UI Automator**，各个框架可能在具体应用上有些不同，如有些偏稳定性，有些适用于**Web**应用，有些能支持跨应用，等等，但其主要思想是通过控件的位置、名称、属性等获取控件对象，并且对控件对象或者坐标模拟用户操作，测试同学通过这些控件的操作和状态变化来完成自动化测试的执行。图2-4里罗列了目前常用的测试框架，也是本书中实践的重点，后面的章节会详细讨论这些框架的原理及应用。



图2-4 目前常用的测试框架

这一小节讨论的自动化测试框架，是在实际项目中总结出来的且基本能运行的通用基础框架原型，它包括三个核心部分：一是如何获取坐标/控件并操作控件模拟用户端事件，二是脚本中的结果如何判断，三是测试结果报告的输出与展示。不管测试人员使用的是Robotium还是其他框架，万变不离其宗，通用原理都可直接图解为图2-5所示的这几个模块。



图2-5 Android自动化测试基本框架模块

1.动作执行

自动化测试的首要条件是能够操作控件，最好像开发同学一样操作控件，如何实现呢？一种最常见的脚本录制方法，其主要思想是记录控件的坐标位置和发生的事件，通过回放脚本完成测试事件流，像Monkey Runner框架就提供比较方便的录制回放功能；另一种方法就是通过工具（比如：源码、UIAutomatorviewer等）获得测试界面的控件

布局，找到目标空间的ID、名字、描述或者位置信息。测试框架可以通过这些信息得到控件对象，并对控件对象执行一系列事件操作像Robotium、UIAutomater等，这个阶段理解为测试的动作执行。

当然对于有跨应用App的控件操作会受到Android进程安全限制，这对于跨应用的操作是一个难点，举个简单的跨应用例子，在测试一款App应用时，它的某个功能会调起系统相机拍照，那这个功能就会涉及跨应用了。像Robotium就无法调用系统的一些INPUT事件完成跨应用的控件操作（其实Robotium从Android 4.3之后开始支持UIAutomation框架，理应可以支持跨应用的），基于Robotium框架的测试脚本跟被测对象需在同一个App或者可以相互访问，一般要求重新签名打包。所以在选定框架时就需要考虑相关的权限问题，当前可以直接支持跨应用的框架有MonkeyRunner、UIAutomater等，后面的章节会详细给出主流框架的分析和使用建议，这里不再细述。

2.结果判断

自动化测试中非常重要的一个环节就是测试步骤的验证，如何能不进行人工干涉而去正确并且自动地检查验证点，这是自动化测试环节中的一个关键点，有些可以借助测试框架的截图对比（像MonkeyRunner）直接完成结果输出，但在实际项目中会有很多具体业务，验证点会比较困难，不同的案例可以用拆解和辅助的验证方式来

完成。比如播放器播放视频时，如何验证视频播放成功，这里就需要设定很多验证点。若播放的时间 ≥ 0 ，可以直接获取video标签里的current time来判断；若是功能测试，则有两种情况：①播放时有画面，可以直接截屏获取；②播放时有声音，可以获取声卡捕捉声音的数据，分析波形文件，但实现起来难度就比较大，后面的章节中有具体的案例介绍。基本结果验证的方法概括如下。

(1) 截图对比。对于GUI的测试，一般会采用截图方式，但最简单的截图也需要考虑到很多附带的条件，大部分框架会有此功能（没有的话也不要紧，可以直接用系统自带的screencap或者其他工具配合使用），但它每秒能截取多少张图片？能否满足我们需要的图片判断条件？截取的图片如何做对比，对比的参照物是什么？做性能测试时，我们的工具是否会影响性能？要不要选择拍照工具而不影响性能？获取图片的速度能否满足性能测试要求？对比的参照物又如何设定？这些信息在每个具体的案例中又会衍生一系列新的问题。就一个具体的案例来说，浏览器打开网页的首字响应时间测试中，对比参照物可以是一张空白的图片，把指定的像素区域与这张空白图片做对比就能判定结果；而浏览器播放视频时的响应时间测试时所截取的图片，就不能这样判断，因为很多视频会在播放之前就显示一个关键帧的图片。为什么会这样？本书后面的章节会针对一些典型业务进行由浅入深的案例分析与工具设计，会用到基本框架但不局限于此，还需

要测试人员更多的对工具和框架的配合应用与改造能力，完成系统性的业务测试需求。

(2) 控件对比。Android UI自动化测试最直接面对的是应用程序界面，界面上存在按钮（`button`）、文本框（`textView`）等，我们都称之为控件。有些应用程序的执行逻辑直接体现在控件的状态显示上，如按钮状态的变化、文本的改变等。常用的自动化测试工具Robotium和UI Automator都提供了获取应用控件的接口，当执行完一定的自动化测试逻辑后，可以将获取控件上的信息与预期的信息进行对比，判断测试结果是否通过。比如测试一个计算器程序，执行3乘以5的测试用例后，可以在结果输出框得到输出的值（`result`），用框架的结果与预期的结果做对比：`assertEqual(result, 15)`，判断这个测试用例的结果是否通过。

(3) 日志分析。日志分析可以作为一个辅助结果判断，比较适合在集成测试阶段做的一些接口、稳定性和性能测试，因为这个阶段产品的日志一般处于打开状态，测试同学可以很方便地收集到日志关键字信息。比如在进行稳定性测试时，可以直接把所有日志输出至文件，在测试执行程序中加入简单的文本处理，对日志信息进行分类检索，像在Java层出现crash时，可以搜索日志中的FATA EXCEPTION/ANR等关键字，提取后面的几十行日志信息进行筛选处理。日志使用的更高进阶是日志埋点，对需要测试的接口埋入上报信

息，一旦测试执行（用户使用）到该点，启动上报接口（或者SDK）送至指定的路径，就可以直接准确地看到测试结果，一般在灰度发布后这些日志信息会被关闭，总体上不会影响产品的功能。但日志的使用也有一些局限性，比如测试同学对日志tag不熟悉，需要先了解代码信息等，还有竞品对比测试时因为拿到的竞品包都是混淆的正式发布包，没有关键日志信息输出，可能需要用其他的手段来判断结果，比如控件对比（当然高手例外，他们可能会使用逆向工程，反编译再注入需要的log）。

3.报告展示

报告展示一般是指给出整个测试的结果信息汇总并进行简单的分析，测试结束后直接输出预警和初步的数据报告，以邮件或者其他形式直接周知项目参与人员。如果执行较大的项目测试，就可能需要多维度、多指标地对待测应用的测试数据进行整合，要求测试同学对平台有较高的设计能力，比如平台需要实现产品中核心业务的性能benchmark测试结果及分析，BVT（Build Verify Testing）测试中发现的问题预警，功能自动化测试脚本中的BUG自动提单，等等。有些自动化测试框架里也带有简单的测试管理平台，根据业务类型决定这些平台是否能满足需求，有的产品需要平台支持多业务的横向与纵向对比信息、竞品的评分信息等，这时就需要做二次开发，形成一个系统性的测试管理工具，也即前文中说的测试的管理平台框架。

2.2 移动终端自动化测试应用场景

移动应用的特点是快速迭代、快速发布，基于移动应用的测试就逐渐转变为轻量测试、灰度发布机制、众测、后台云控系统等，甚至有损发布，很多类似的产品不断涌现，随着市场和用户日趋成熟，产品的功能差异性不大，那么竞争的关键就转移至品质与口碑，就要求研发团队严把质量关卡，赢得用户口碑。在这样的前提下，需要在质量团队引入更强大的平台或者用测试方法支撑业务的品质，要求自动化测试的思路深入至品质的各个角落，以真正为产品质量服务。

不妨梳理一下在应用App测试的时候，需要在哪些耗时耗力以及特殊要求的场景进行自动化测试。读者首先可能会想到兼容性测试，因为众所周知的Android碎片化问题，不同的机型适配问题可能五花八门，这里就需要有合适的兼容性方案去尽可能地覆盖测试，在实际项目中也确实是这样。这里，和读者一起来梳理一下移动平台自动化测试到底能做什么，哪些场景更适合用自动化测试，这些自动化测试是否跨越了我们传统的误区。图2-6所示为笔者梳理的Android自动化测试应用场景，每个领域里内容不是非常齐全，目的是引导读者一起去思考如何完善质量保证体系的自动化测试场景。

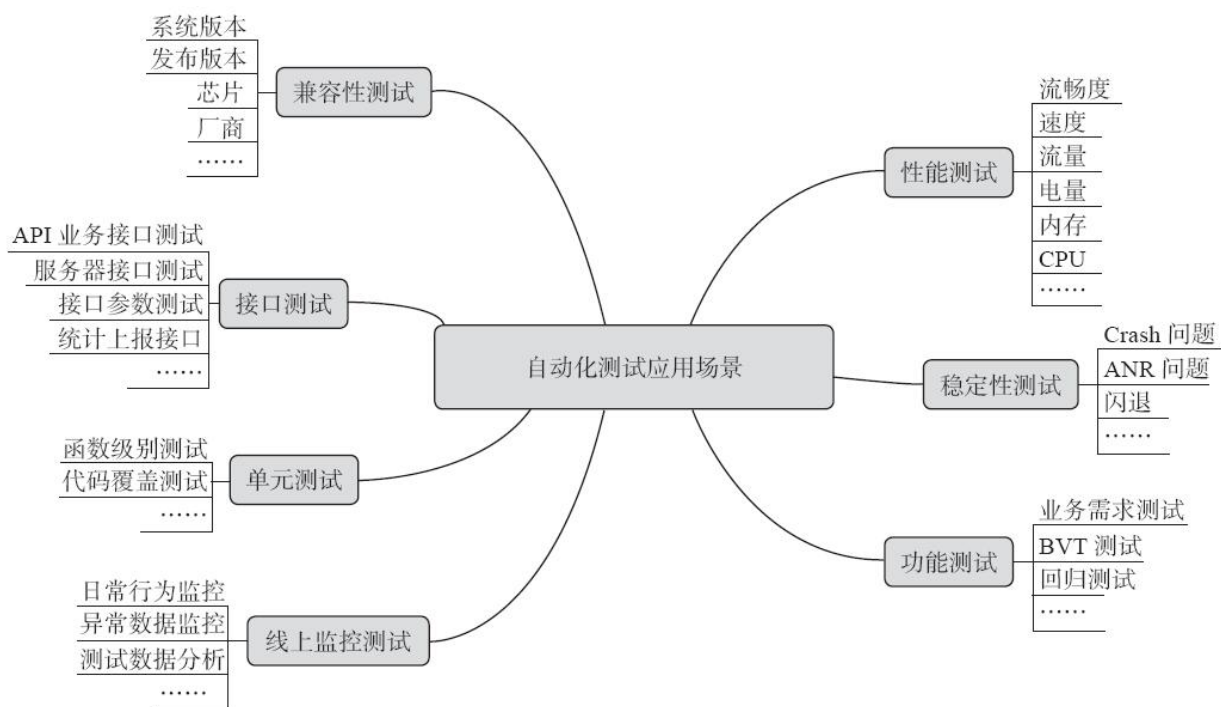


图2-6 Android自动化测试应用场景

(1) 性能测试。移动终端应用，不管是Native还是WebView的应用，对性能要求都非常高，主要是卡顿、耗电、速度这几个常见关键性的指标，而这类测试重复性强，指标路径固定，并且质量指标中又需要分为横向与纵向对比情景，等等，形成一个庞大的测试矩阵，自动化测试支持才能更快捷地完成测试任务，一般性能测试会考虑选用自动化方案，此方案非常适合性能测试。

(2) 稳定性测试。Android平台一般都会联想到用系统自带的Monkey工具进行测试，此工具既易上手也实用，但运用起来有非常多的讲究和技巧，简单的Monkey工具不一定能完成使命，在测试中也需

要花费心思去对原生的Monkey进行改造，以满足不同业务的稳定性测试需求。

（3）功能测试。关于功能测试的争议比较多，因为产品都需要快速迭代，而脚本的稳定性、实现时间等成本开销大，真正发挥作用也需要不断地打磨，并且还有很多后期维护成本，所以比较折中的办法是做一些BVT测试和持续集成配合，在开发编译新的build后直接运行这些核心的BVT用例，以免出现严重的Regression/Block问题，日常的工作中选定较小范围的用例及适合的框架一般就可以解决问题。

（4）兼容性测试。不同的业务可能会有不同的适配要求，现在比较常用的方法是直接使用业界比较成熟的测试平台，如Testin、百度MTC、腾讯优测平台等，一般情况下平台能提供几百甚至上千台机器进行测试。本书后面也给出了兼容性测试方案，供各读者在项目中实践。

（5）接口测试。这块的测试主要是集中一些重要的API测试，和PC端的接口测试思路一样，都是通过脚本去遍历所有重要的参数等，并且抛开界面的干扰快速测试以至稳定。像浏览器里常见的就有JS API接口测试，当然这块可能需要开发同学的接口定义文档或者口头支援，梳理业务的关键API和参数列表以及相应的依赖关系等，是非常适合用自动化测试去实现的，脚本也相对简单稳定，而且效果明显。

(6) 单元测试。Android终端用Android Junit可以快速方便地实现单元测试。很多公司单元测试工作都是由开发同学自行完成，但在移动互联网时代，基于敏捷开发测试前移的大环境，部分测试同学也会直接参与单元的编写和执行，比如，腾讯Tencent OS（TOS）项目团队就是由测试同学进行单元低层OS系统的单元测试，后面的章节也会讲到这块的测试案例。

(7) 线上监控测试。这块测试方向不应该直接归属于传统的自动化测试范畴，因为它不需要常规情况下提到的自动化测试框架支持，也不需要开发测试用例脚本，这里主要是对线上测试数据的监控，并且利用大数据分析进行“自动化”测试，在互联网产品中极为适用而且能非常直接地体现产品的质量。举个简单的例子，通过浏览器的网页浏览功能，可以监控用户在浏览网页时有多少个浏览失败的网站、是否会出现必然浏览失败的网站、出现浏览失败的网站的地域/DNS是什么等，如此层层过滤，最后得到的关键信息会直接指导测试人员缩小测试范围，提高测试效率。但本书中没有准备这方面的案例，笔者在梳理这块内容时，为了保证知识的完整性在这里做一个小的铺垫，本书再版时会添加这方面的案例。

2.3 本章小结

其实自动化测试只是日常测试中的一个辅助手段，说白了，它就是利用自动化测试框架以及工具能理解的程序代替人去完成测试，通过比较执行的结果来判断测试是否通过。它的优势很明显，无论多复杂的操作，即使反复执行成千上万遍，只要程序没有问题，它都会快速地执行完毕，而且比人工轻松得多；但劣势也非常明显，最大的缺点就是工具是人工思考之后的一个固化程序，它不会变通，是按照人的旨意来完成相应的任务。所以测试人员对工具的应用和改造能力显得尤为重要，通过对被测应用的深入认识与思考，转化成可以实现的测试需求，再配合这些工具和擅长代码的同事进行封装，可以完成测试中各个领域的人工替代工作。本章对移动终端App应用的开发、测试和上线各个阶段需要的测试进行自动化测试梳理，分析自动化测试的应用场景，为后面的框架内容和案例提前做铺垫。

第3章 Robotium框架工作原理及实践

2010年，当Android还处于发展早期时，拥有丰富自动化测试经验的Renas Reda创建了Robotium项目，在Robotium发展到4.0版本时开始支持App中的Web自动化，经过几年的发展，Robotium现在已经是一款成熟、全面、稳定的自动化测试框架。更重要的是，Robotium是一款开源的测试框架，在世界各地都有活跃的贡献者对其进行更新与维护，因此，无须担心将来Robotium会随着Android的发展而变得不可用、不易用，相反，Robotium每天都在变得更加强大。

任何技术都离不开基础知识。首先，本章将介绍Robotium是什么以及有关Robotium的一些基础知识，让读者了解Robotium的基本规则。其次，将从Native和WebView两方面简析Robotium测试框架的运作原理，并介绍Robotium的实际应用以及笔者在实践过程的一些经验技巧，以加深读者对Robotium测试框架的理解。最后，本章选取一般项目中常见的一些场景介绍如何使用Robotium解决实践中的问题。

本章知识结构图如图3-1所示。

阅读完本章后，读者应该能比较全面地了解Robotium测试框架并知道如何使用了，由于本章只介绍Robotium相关知识，关于Robotium在项目方面的实际应用则可阅读第10章。

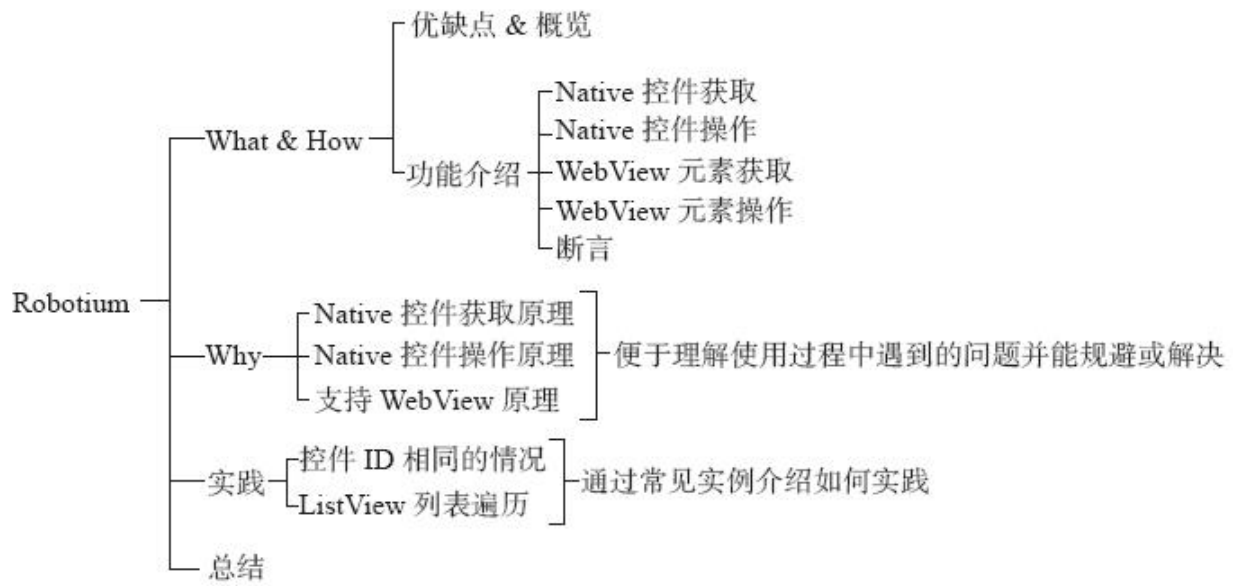


图3-1 本章知识结构图

3.1 Robotium常用功能

3.1.1 什么是Robotium

Robotium是一款类似Selenium但面向Android端的开源自动化测试框架，既支持测试Native应用，也支持测试Hybrid应用（混合模式应用，指介于WebApp与NativeApp两者之间的App，兼具Native App良好的用户交互体验的优势以及Web App跨平台、易变更的优势）；既支持黑盒形式的自动化测试，也支持白盒形式的自动化测试。通过Robotium用户可以编写出更强大健壮的UI自动化测试用例，并可以应用在功能测试、系统测试、用户验收测试等多种测试场景中。

Robotium主要具有以下优势：

- 同时支持Native应用和Hybrid应用。
- 由于是基于Instrumentation的测试，测试代码运行于被测应用所在的进程，控件识别与模拟UI事件都可以快速执行，因此测试用例执行速度更快。
- 由于是通过在运行时识别控件而非通过固定坐标方式，因此测试用例可以更健壮。

- 由于支持黑盒方式，不需要深入了解被测应用即可开展测试，因此编写用例花费的时间可以更少。

- 由于可以通过Maven、Gradle或者Ant运行测试用例，因此可以很好地作为持续集成的一部分。

Robotium缺点：

- 由于是基于Instrumentation的事件发送，因此无法跨应用。
- 代码运行在被测进程，可能影响被测进程的内存、CPU占用，若用于性能监控数据会有误差。

注：项目开源地址：<https://github.com/RobotiumTech/robotium>

3.1.2 Robotium提供的类

Robotium对外主要提供以下几个类：

- By: Web元素的选择器。
- Condition: 接口类，用于等待。
- RobotiumUtils: 工具类。
- Solo: 对外提供各种API。
- Solo.Config: Solo配置类。
- SystemUtils: 系统级工具类。
- TimeOut: Solo配置类。
- WebElement: Web元素的抽象类。

其中Solo类是主要对外提供各种API的类，Solo类采用中介者模式，持有com.robotium.solo包下的其他类的实例对象，当我们调用Solo类中的API时，大多数是转而调用com.robotium.solo包下其他类的方法。com.robotium.solo包下主要有以下类：

- Getter: 提供控件获取相关API。
- ActivityUtils: 提供Activity相关API。
- Asserter: 提供断言相关的API。
- Clicker: 提供模拟点击相关的API。
- ScreenshotTaker: 提供截图相关的API。
- Scroller: 提供滚动相关的API。
- Searcher: 提供控件搜索相关的API。
- ViewFetcher: 提供控件过滤相关的API。
- Waiter: 提供控件等待相关的API。
- WebUtils: 提供Web支持相关的API。

Robotium为了简化测试用例的编写，将以上的这些类都置为protected，对外只提供Solo类，因此，在编写测试用例时，主要实例化Solo类即可，本章介绍的API默认也均为Solo类中的方法。

3.1.3 环境搭建

使用Robotium进行自动化测试的工程采用的是Android Junit Test工程，基础环境与Android开发环境一致，为了方便本书第3章及第10章的讲解，本书的官网（<http://tmq.qq.com/>）附上了改造后的NotePad和NotePadTest工程，环境搭建步骤如下：

步骤1：安装基础环境（搜索引擎搜“Android开发环境搭建”）。

步骤2：导入工程。

下载随书官网中的NotePad和NotePadTest两个工程，打开Eclipse → File → Import，选择“Existing Projects into Workspace”，如图3-2所示。选择NotePad工程所在的目录，导入NotePad工程。使用相同的步骤，导入NotePadTest工程。如图3-3所示。

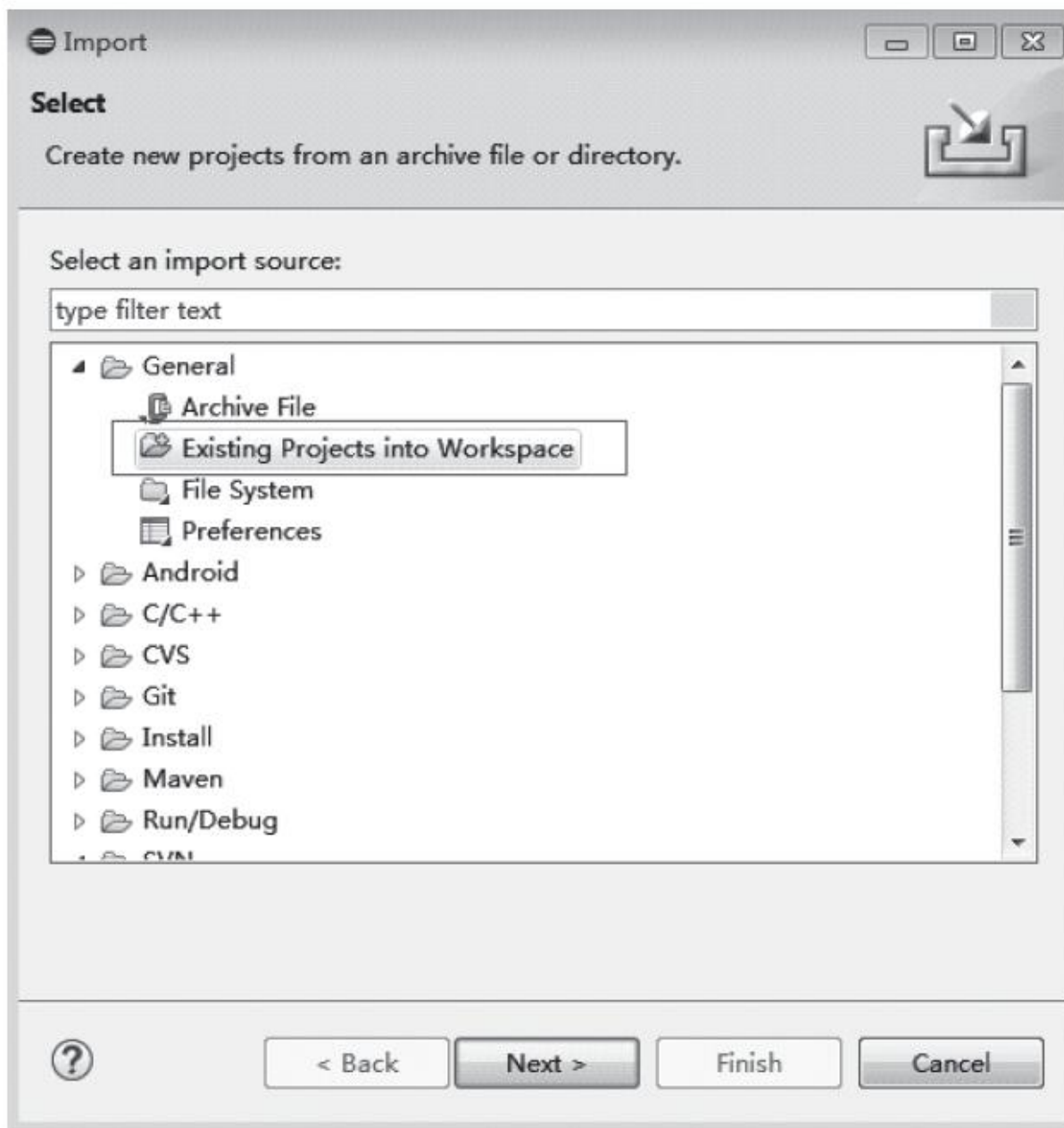


图3-2 选择导入已存在的工程至工作空间

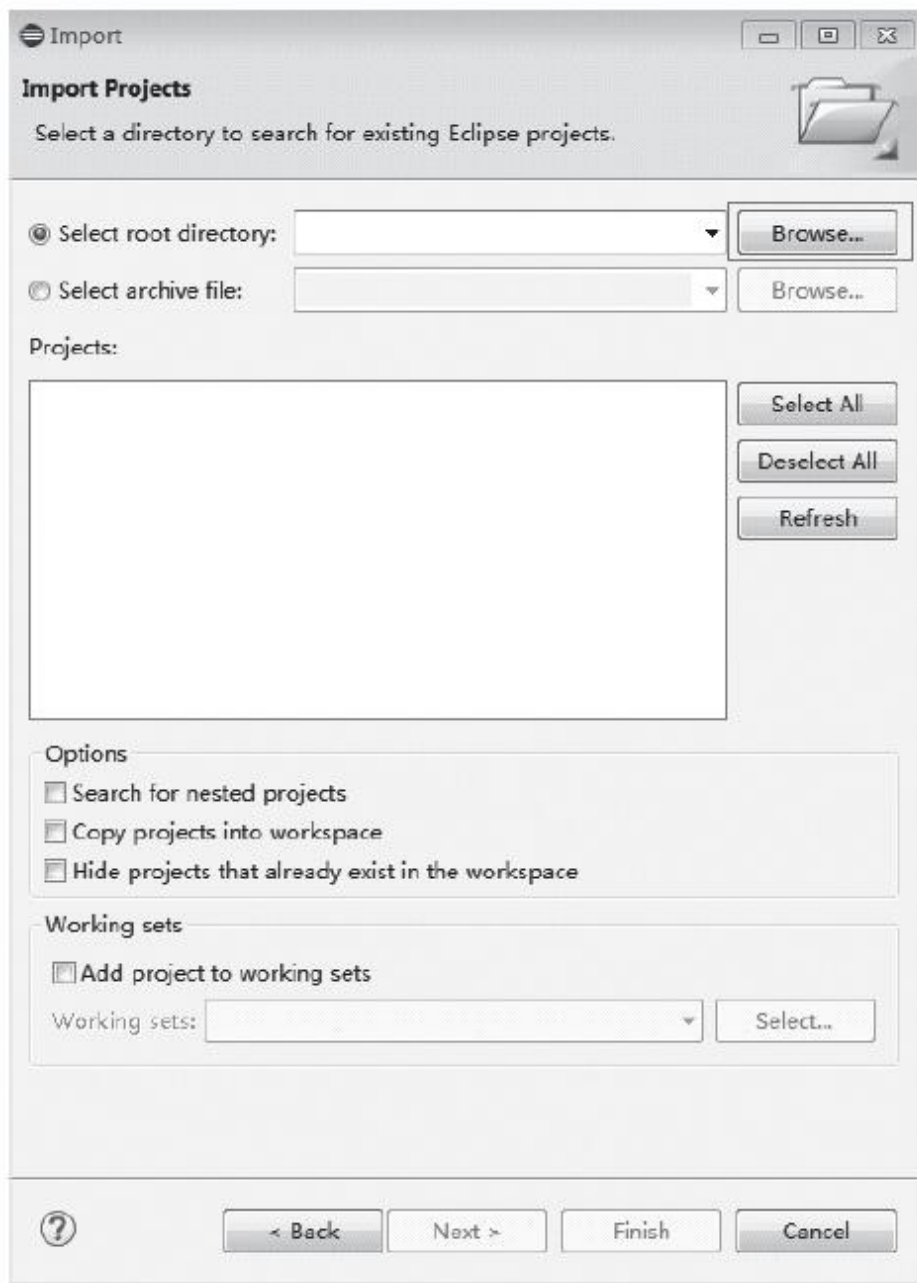


图3-3 选择NotePad及NotePadTest工程所在的目录

步骤3: 配置工程。

导入两个**Demo**工程后，由于两个工程均包含有一些配置文件，因此如果没有提示错误，则可以直接使用；如果还有提示错误，请依实

实际情况检查以下配置项。

(1) 配置签名：两个工程需要签名一致，这里使用Android开发环境中自带的debug.keystore进行签名，因此需要确保环境中包含该签名文件，签名配置查看：Window → Preferences → Android → Build，如图3-4所示。

(2) 配置build target：build target即指定使用哪个Android平台来构建这个项目，两个工程均配置为使用target=android-19，即使用sdk中platforms目录下android-19目录中的android.jar这个jar包编译项目，如图3-5所示。

若你的开发环境中未下载相应的API level的jar包，请使用SDK Manager下载，或者自行将project.properties配置文件中的target换成用户机器中已下载的API level。

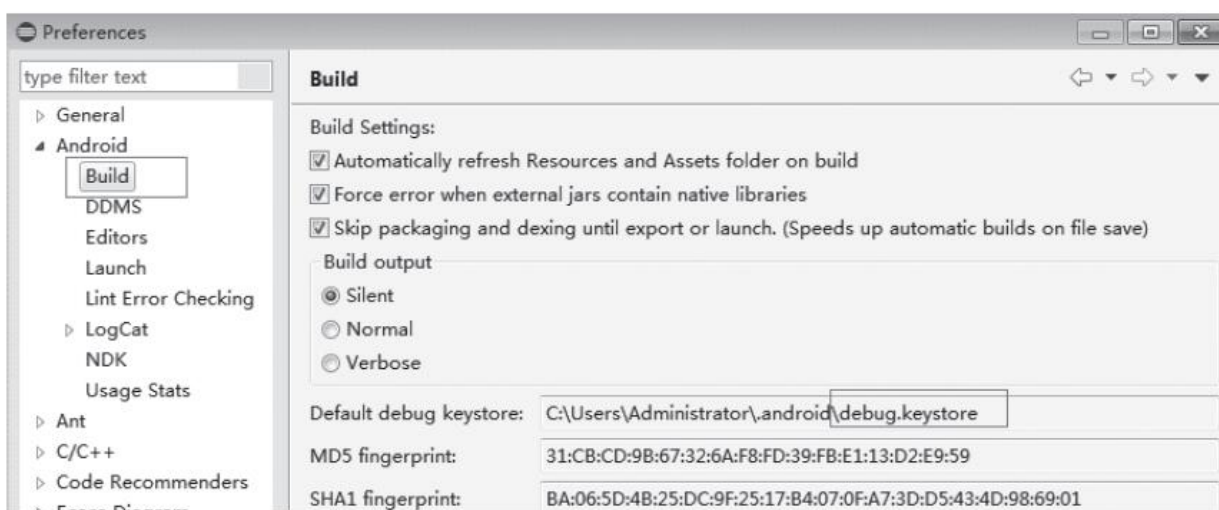


图3-4 检查签名文件

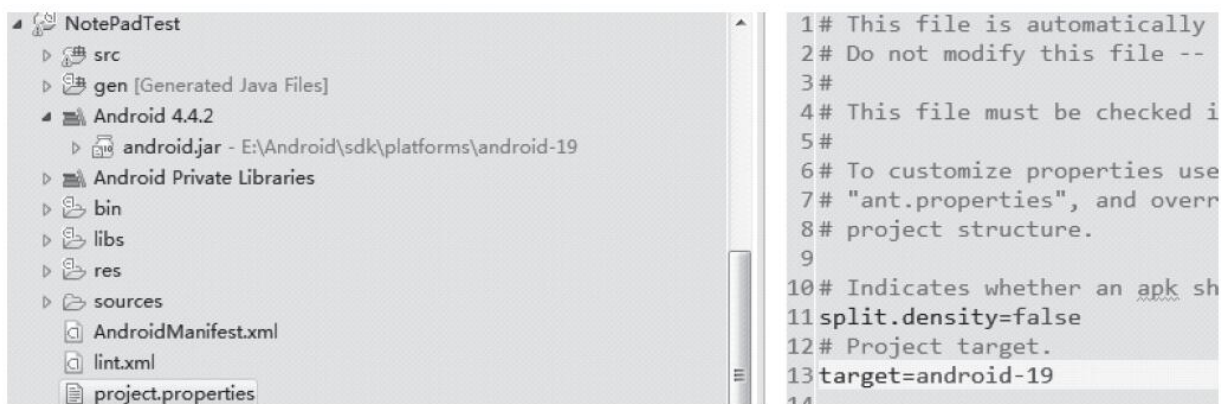


图3-5 配置build target

(3) 引用Robotium的jar包并关联源码：Robotium测试框架以jar包形式提供，在测试工程中引用Robotium的jar包即可。Android的Junit形式测试工程与Android工程一样，将要引用的jar包放入libs目录下，在Eclipse中将默认变成Android Private Libraries私有库，这样默认在Eclipse中就可以引用该jar中的API，在编译时也会将其编译进测试工程的APK中，如图3-6所示。

此外，为了方便查看Robotium中的源码实现，一般也会选择关联引用的jar包的源码，如上图所示，在libs中新建相应的properties文件，然后使用src=形式的命令指定源码所在的目录即可。

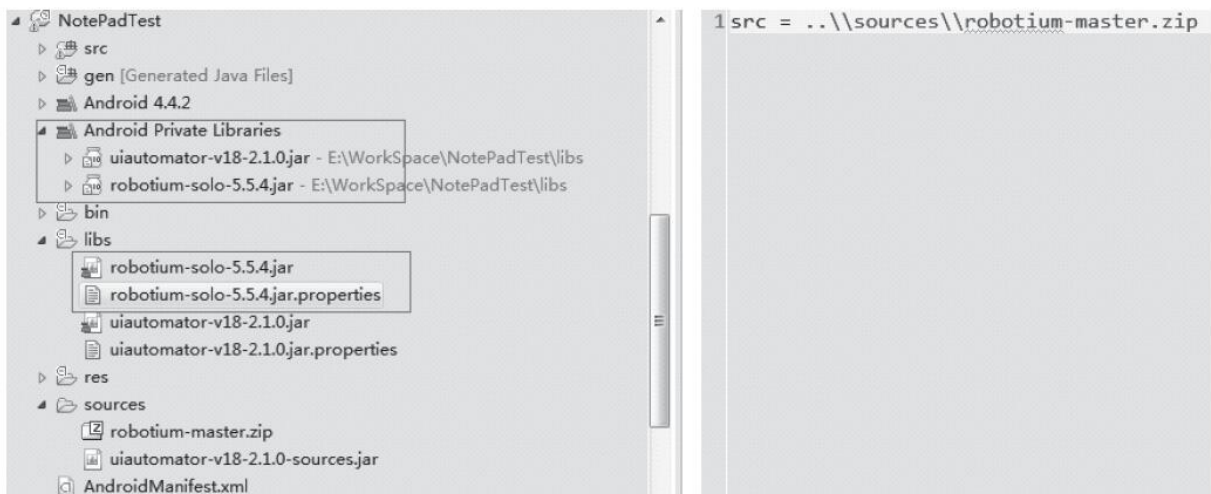


图3-6 配置Robotium的jar包并关联源码

(4) 配置编码：新导入工程后，由于工程里会含有一些中文注释，常常会由于编码不一致，导致代码结构被破坏而引起工程编译出错，Demo中的两个工程均采用UTF-8编码，因此需要检查导入后编码是否为UTF-8，右键工程依次选择Properties→Resource，查看编码，如图3-7所示。

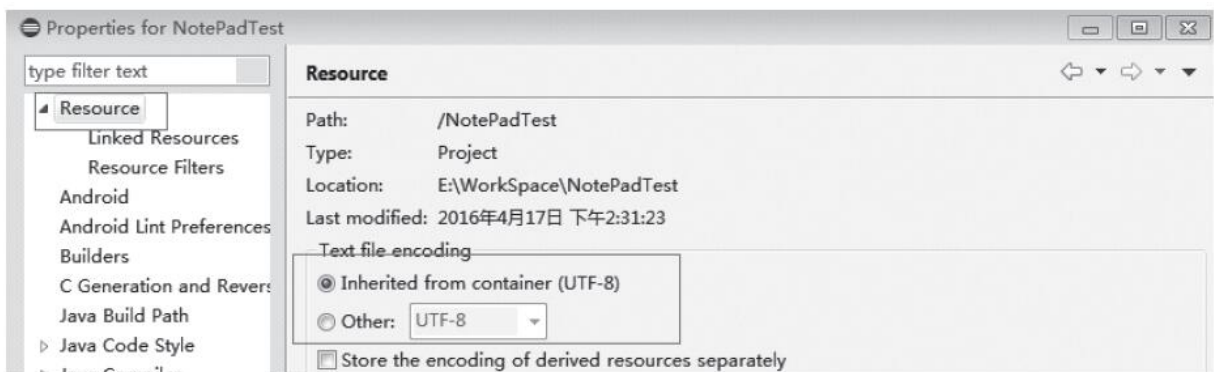


图3-7 配置编码

(5) 配置Instrumentation: 测试工程需要配置Instrumentation以指定要注入的被测进程, 即指定被测App, 在NotePadTest中使用<instrumentation>标签指定targetPackage为被测应用的包名, 如图3-8所示。

步骤4: 运行示例。

首先确保有手机开启了USB调试, 并连接了电脑, 然后如图3-9所示, 右键选择示例的测试类, 例如右键选择NotePadTest.java类, 选择Run As → Android Junit Test, 即可运行Demo中的测试用例。

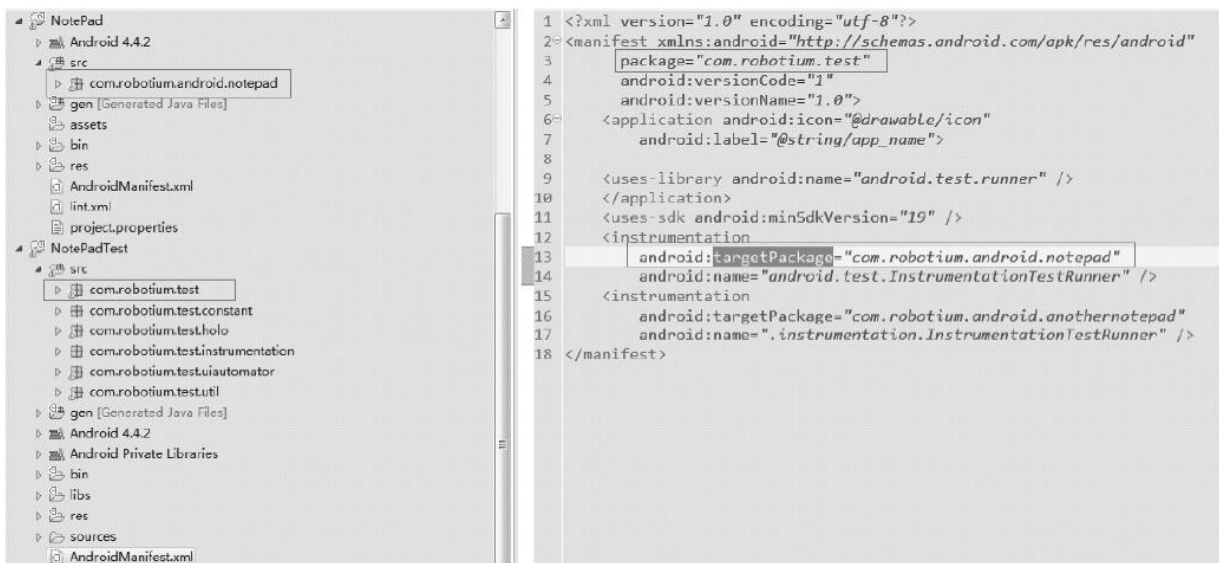


图3-8 配置Instrumentation

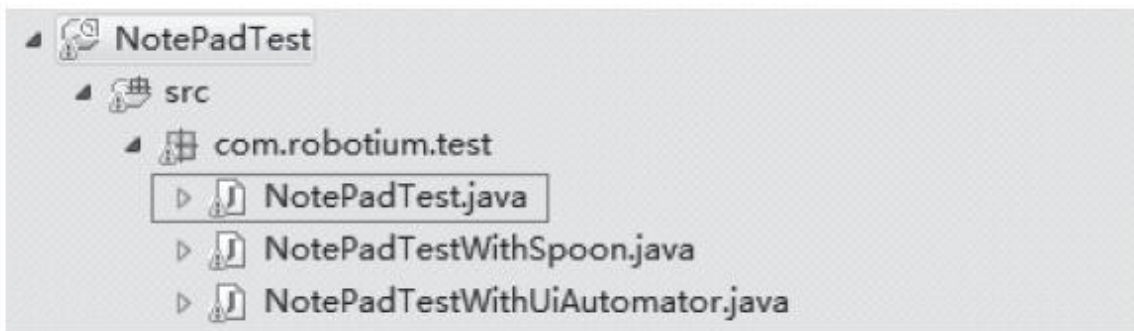


图3-9 运行示例

对于环境搭建，新手较容易出现如下问题：

常见问题1：The import android cannot be resolved

需要检查上文配置工程部分中配置的build target是否正确。

常见问题2：java.lang.NoClassDefFoundError:

com.robotium.solo.Solo

NoClassDefFoundError指在编译时该类是存在的，但在运行的时候找不到该类，报找不到Solo类时一般意味着Robotium的jar包未打进测试工程的apk包中。首先，右键测试工程→Build Path→Configure Build Path，查看确保在Libraries中包含了Robotium，如图3-10所示。由于demo中将Robotium的jar包放至libs目录下了，因此默认将包含至Android Private Libraries中。

然后，如图3-11所示，在Build Path的Order and Export中确保Robotium的jar包处于勾选状态（处于勾选状态即意味着该jar的Class类将被打包进测试工程的APK中，而例如Android SDK中的android.jar包，由于其Class类在手机的Android系统中已存在，因此不需要勾选，在运行时也可以找到相应的类）。

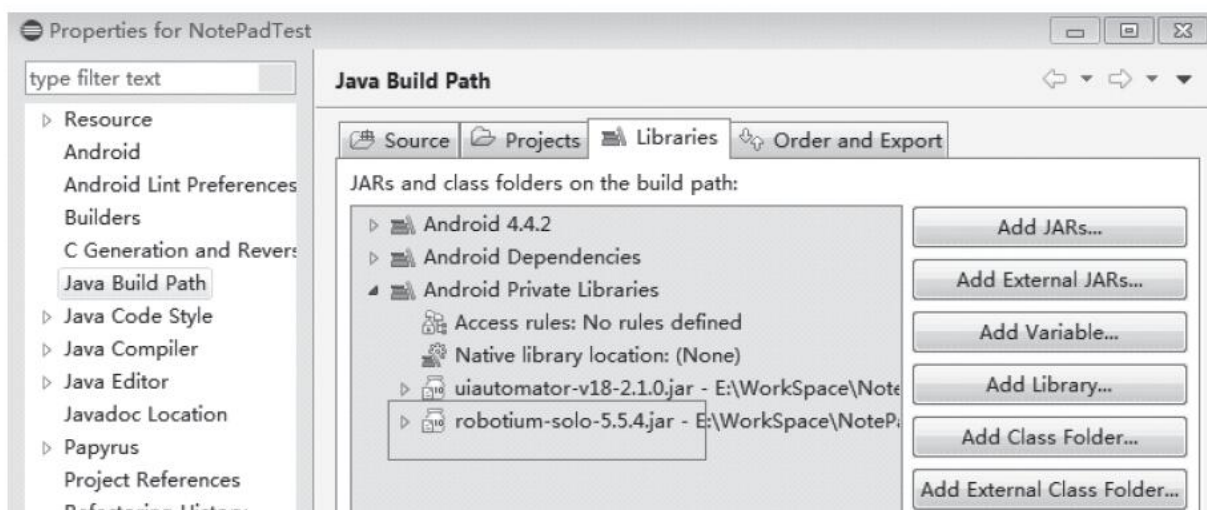


图3-10 确保Libraries中包含Robotium

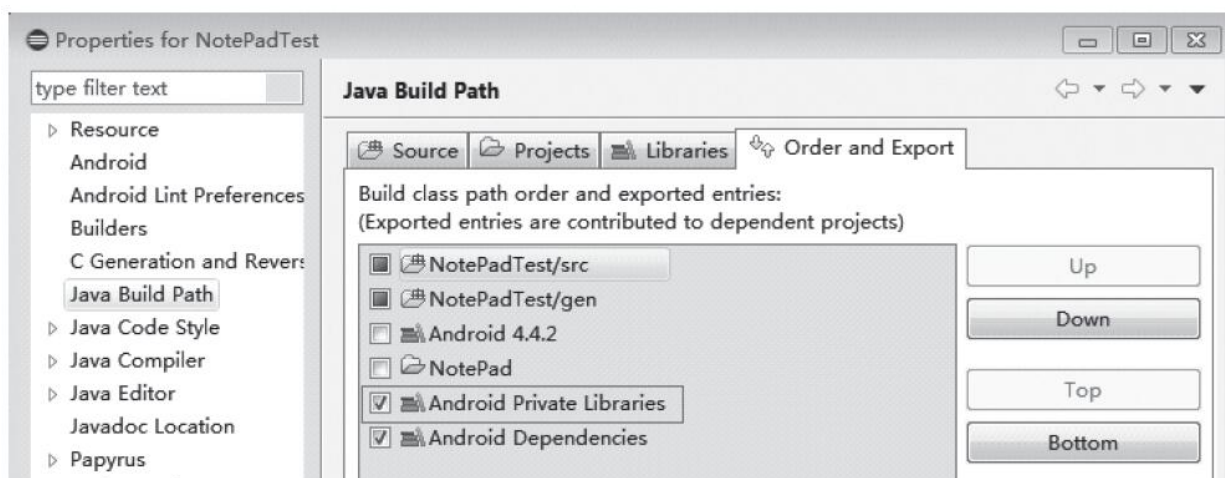


图3-11 确保Order and Export中Robotium的jar包被勾选

3.1.4 Robotium的控件获取、操作及断言

Robotium是一款在Android客户端中的自动化测试框架，它需要模拟用户操作手机屏幕。要完成对手机的模拟操作，应该包含以下几个基本操作：

- (1) 需要知道所要操作控件的坐标。
- (2) 对要操作的控件进行模拟操作。
- (3) 判断操作完成后的结果是否符合预期。

因此，本节将从控件获取、控件操作及操作后断言来介绍Robotium，此外，由于WebView在控件获取和控件操作上都与Native完全不同，将对其做单独介绍。

1.Native控件获取

从Robotium中获取Native控件主要有两大方式：一个是根据被测应用的控件ID来获取；另一个是先获取当前界面所有的控件，对这些控件进行过滤封装后再提供相应的获取控件的API。

- 1) 根据被测应用的控件ID来获取

根据控件ID获取见表3-1。

表3-1 根据控件ID获取

返回值	方法及说明
View	getView(int ID) 根据 ID 获取控件
View	getView(String ID) 根据 ID 获取控件

根据String型ID获取控件:

```
ImageView mIcon = (ImageView) solo.getView("mypic");
```

在Android中，所有的控件都继承自View，因此，如果被测应用中的控件有唯一ID的话，就可以使用这种通过ID形式唯一获取所要操作的控件。

例如获取RelativeLayout或LinearLayout:

```
RelativeLayout rel = (RelativeLayout) solo.getView("example1");  
LinearLayout lin = (LinearLayout) solo.getView("example2");
```

由于Android中所有的控件都继承自View类，而对于开发人员的自定义控件，这些自定义控件也基本是继承自Android的基础控件扩展而来的，因此通过这种方式几乎可以获得所有类型的控件，获取相应类

型的控件时只要进行转义即可，因此当控件拥有唯一ID时，推荐使用该方式。

控件ID可以通过Android SDK中提供的工具来查看，例如%ANDROID_HOME%\tools\uiautomatorviewer.bat工具，在Android 4.3及以上系统版本的手机上，可直接查看到UI界面的ID。

2) 根据控件类型的索引、文本来获取

根据文本获取见表3-2。

表3-2 根据文本获取

返回值	方法及说明
Button	getButton(int index) 根据 index 索引获取控件
Button	getButton(String text) 根据文本 text 获取控件

此方式是Robotium先将当前界面中的所有控件全部获取，然后按控件类型、索引进行过滤后再获取指定的控件View。

根据index索引获取控件：

```
//返回界面中第一个类型为
Button的控件


Button loginBtn = (Button) solo.getButton(0);
```

其他的如`getEditText (int index)` 、`getText (int index)` 均同理。

根据文本`text`获取控件:

```
//返回界面中文本为 '登录' 类型为  
Button的控件  
  
Button loginBtn = (Button) solo.getButton("登录  
");
```

其他的如`getText (String text)` 、`getEditText (String text)` 均同理。

 **提示** 在Robotium中查找控件时，如果找不到相应ID或文本的控件，测试框架会throw出“View with id×××is no found”或者“with text×××no found”等Throwable异常，若我们并不希望因此而报错，则可以使用try catch Throwable来捕获。

3) 根据控件类型进行过滤

根据类型过滤见表3-3。

表3-3 根据类型过滤

返回值	方法及说明
ArrayList<View>	<code>getCurrentViews()</code> 获取当前界面或弹框中所有的控件
ArrayList<T>	<code>getCurrentViews(Class<T> classToFilterBy)</code> 获取当前界面或弹框中所有控件类型为 <code>classToFilterBy</code> 的控件
ArrayList<T>	<code>getCurrentViews(Class<T> classToFilterBy, View parent)</code> 获取父控件 <code>parent</code> 下所有控件类型为 <code>classToFilterBy</code> 的控件

获取当前界面或弹框中所有控件类型为**TextView**的控件:

```
ArrayList<TextView> allTextViews = solo
    .getCurrentViews(TextView.class);
```

获取指定父控件下所有控件类型为**TextView**的控件:

```
RelativeLayout rel = (RelativeLayout) solo.getView("example1");
ArrayList<TextView> allTextViews = solo
    .getCurrentViews(TextView.class, rel);
```

同样是过滤出指定的控件类型，不过该方法是从父视图`parent`中开始过滤，当不指定`parent`，即`solo.getCurrentViews (TextView.class, null)`时，则和`solo.getCurrentViews (TextView.class)`一样，返回的是当前界面中所有的。

移动App一般节奏很快，UI布局结构也经常随着功能的变更而变动，例如“登录”按钮从最上面变到了最下面，因此通过索引或文本来获取控件是有很大大隐患的。很多时候，通过巧妙地控件过滤可以更准确地找到相应的控件。

2.Native控件操作

对于Android端的自动化测试而言，当我们获取到期望的控件后，接下来就是对该控件进行点击、长按、文本输入、拖动等模拟操作。除此之外，UI自动化测试为了贴近用户的真实使用及自身健壮性，还需要时延等待、页面加载等待；为了判断界面是否符合预期，则还需要控件搜索、界面截图等操作。

1) 点击、长按操作

点击长按见表3-4。

表3-4 点击长按

返回值	方法及说明
void	clickOnView (View view) /clickLongOnView (View view) 点击指定的 View 控件 / 长按指定的 View 控件
void	clickOnScreen (float x, float y) /clickLongOnScreen (float x, float y) 根据坐标 x, y 点击屏幕 / 根据坐标 x, y 长按屏幕

Robotium是基于控件的自动化测试框架，当获取到要操作的控件后，直接对控件进行点击、长按或文本输入等操作即可。

点击指定的View控件：

```
Button loginBtn = (Button) solo.getView("loginBtn");
solo.clickOnView(loginBtn)
```

Robotium还提供了点击文本、点击图片的API，例如clickOnText (String text) 、click-OnButton (String text) 等，这类API类似于前文

所介绍的，先根据文本获取控件，再发送点击事件：

```
Button loginBtn = (Button) solo.getButton("登录");
solo.clickOnView(loginBtn)
```

类似于点击、长按指定的View控件：

```
Button loginBtn = (Button) solo.getButton("登录");
solo.clickLongOnView(loginBtn)
```

需要注意的是，Robotium的点击事件是通过Instrumentation发送的，因此该类点击方法不能点击非被测应用的区域，例如不能点击至通知栏所在的区域，否则会出现类似如下的异常：

```
java.lang.SecurityException: "Injecting to another application requires
INJECT_EVENTS permission"
```

因此在使用Robotium编写测试用例时，需要注意其无法跨应用的缺点，从而尽量避免出现此场景，有些场景偶然性地无法规避，可以采用try catch Throwable的形式捕获异常，而对于需要跨应用的场景，则可以使用9.4.2节介绍的UI Automator结合Instrumentation模式进行处理。

```
try {
    } catch (Throwable e) {
    }
```



技巧

在手机设置-开发者选项中，可以开启“指针位置”，开启“指针位置”后，再触摸屏幕时，可实时显示屏幕坐标。调试时为了更准确地知道对屏幕的什么地方进行了操作，也常常同时开启“显示触摸操作”开关。

2) 操作输入框

操作输入框见表3-5。

表3-5 操作输入框

返回值	方法及说明
void	<code>enterText(EditText editText, String text)</code> 在指定的 EditText 中输入文本 text
void	<code>typeText(EditText editText, String text)</code> 在指定的 EditText 中键入文本 text
void	<code>clearEditText(EditText editText)</code> 清空指定的输入框

在自动化测试过程中，当我们可以准确获取控件，并能模拟点击、长按等基本操作后，就可以在被测应用中进行自由跳转，然后可能就需要进行一些输入操作。测试框架中主要提供了`enterText`

`(EditText editText, String text)`和`typeText (EditText editText, String text)`两种方法，前者直接对`EditText`文本框进行赋值，不会有文本输入的展示过程，而后者则会一个文本一个文本地输入，更贴近真实用户的操作。

```
EditText userET = (EditText) solo.getView("example_et_id");
solo.enterText(userET, "my_user_name") //直接对文本框赋值
```



```
solo.typeText(userET, "my_user_name")           //会展示输入的过程
```

3) 滑动、滚动

滑动、滚动见表3-6。

表3-6 滑动、滚动

返回值	方法及说明
void	drag(float fromX, float toX, float fromY, float toY, int stepCount) 从起始 x,y 坐标滑至终点 x,y 坐标；通过 stepCount 参数指定滑动时的步长
void	scrollToTop() / scrollToBottom() 滚动至顶部 / 滚动至底部
void	scrollUp() / scrollDown() 向上滚动屏幕 / 向下滚动屏幕
void	scrollListToLine(AbsListView absListView, int line) 滚动列表至第 line 行

在Android中，常用的操作还有各种滑动手势，如上拉、下拉、左滑、右滑等。在滑动方面，测试框架主要提供了两类支持，一类是根据坐标进行滑动从而可以模拟各类手势操作，另一类则是根据控件来直接进行滚动操作。

根据坐标进行滑动的主要是drag（float fromX，float toX，float fromY，float toY，int step Count），这里的参数包括起始位置的x与y坐标、终点位置的x与y坐标，还有步长stepCount。其中步长stepCount的意思是，假如要从A点滑到B点，如果步长为1，那么将直接产生从A点到B点的手势操作，滑动速度很快；如果步长为100，则将从A到B分成

100等份，例如A、A1、A2...B，然后依次从A滑到A1，再从A1滑到A2、A2滑到A3.....这样滑动更慢但结果也更精确，例如当我们在手机上快速从下往上滑动时，列表滑动是有惯性的，会快速滚动，而这常常不是我们所需要的。

根据控件进行滚动主要有滚动至顶部、底部等方法。scrollToTop

() 方法可以将当前屏幕滑至顶部，如果当前是ListView则滑至列表的顶部，如果是WebView则滑至页面的顶部。同样地，scrollToBottom

() 可将界面滑至底部。类似的还有向下滑一屏的scrollDown () 方法和向上滑一屏的scrollUp () 方法。与前文介绍的drag方法不同的是，这类滚动调用的是相应控件自身的API，例如WebView的滚动调用的是控件自身的pageUp (boolean top) 或pageDown (boolean bottom) 方法。因此，这种方式与drag方式最大的区别在于，drag是实际地模拟手势操作，当上拉时，如果ListView有监听上拉加载更多，那么使用drag是可以触发上拉加载更多的，而scrollUp () 则不能。

4) 搜索与等待

搜索与等待见表3-7。

表3-7 搜索与等待

返回值	方法及说明
void	sleep(int time) 休眠指定的时间，单位毫秒
boolean	searchText(String text) 从当前界面搜索指定文本
boolean	waitForView(int id) / waitForText(String text) 等待指定控件出现 / 等待指定文本出现
boolean	waitForActivity(String name) 等待指定的 Activity 出现
boolean	waitForLogMessage(String logMessage) 等待指定的日志信息出现
boolean	waitForDialogToOpen() / waitForDialogToClose() 等待弹框打开 / 等待弹框关闭

UI自动化测试常常被诟病运行不稳定，除了项目快速迭代导致界面经常变更这一不可控因素外，脚本常常运行出错就是由于没有合适的等待机制导致控件未找到、点击异常等问题，要想测试用例能够快速且稳定地运行，合理使用等待是关键要素之一。

Robotium中提供了诸多与等待相关的API，但是实际情况中需要等待的操作往往要复杂得多，因此测试框架中也提供了Condition模式，即waitForCondition（Condition condition，int timeout）方法，使用该方法时，实现Condition接口并重写isSatisfied（）方法，isSatisfied（）为true时将跳出等待。通过这种模式我们可以自定义实现更多类型的等待操作，如代码清单3-1所示。

代码清单3-1 使用waitForCondition模式实现等待

```
public void waitForAppInstalled(final String appName, int timeout) {
    waitForCondition(new Condition() {
        @Override
        public boolean isSatisfied() {
            sleeper.sleepMini();
        }
    }, timeout);
}
```

```
        return checker.isAppInstalled(appName);
    }
    }, timeout);
}
```

当然了，我们也可以使用超时机制来实现，如代码清单3-2所示。

代码清单3-2 使用TimeOut模式实现等待

```
public void waitForAppInstalled(final String appName, int timeout) {
    long endTime = SystemClock.uptimeMillis() + timeout;
    while (SystemClock.uptimeMillis() < endTime) {
        if (checker.isAppInstalled(appName)) {
            break;
        }
        sleeper.sleep();
    }
}
```

需要注意的是，Robotium中查找控件、点击控件等API都默认使用了搜索与等待机制，当我们使用上文提到的获取控件、点击控件相关操作时，测试框架已经做好了等待操作，因此非特殊情况是不需要额外增加等待操作的步骤的。太多的等待将使用例执行变得缓慢低效，因此在用例编写调试过程中应该做好平衡。

5) 截图及其他

截图及其他见表3-8所示。

表3-8 截图及其他

返回值	方法及说明
void	<code>takeScreenshot(String name)</code> 截图，图片名称为指定的 <code>name</code> 参数，图片默认路径为 <code>/sdcard/Robotium-Screenshots/</code>
void	<code>finishOpenedActivities()</code> 关闭当前已打开的所有 Activity
void	<code>goBack()</code> / <code>goBackToActivity(String name)</code> 点击返回键 / 不断地点击返回键直至返回到指定的 Activity
void	<code>hideSoftKeyboard()</code> 收起键盘
void	<code>setActivityOrientation(int orientation)</code> 等待设置 Activity 转屏方向

自动化测试过程中，因为都是自动化执行的，当用例执行失败时，除了日志外，最方便解决定位问题的就是运行时的截图，有了截图定位问题往往事半功倍，Robotium中提供了单次截图及截取一系列图片的功能。`takeScreenshot()` 方法可以直接截取当前屏幕，并将其默认地保存在`/sdcard/Robotium-Screenshots/`目录下，要更改图片名称则使用`takeScreenshot(String name)`，要截取某时间段内一个序列的话则可以使用`startScreenshotSequence(String name)`。那么如何更好地在自动化中使用截图功能呢？一般情况下我们更希望的是在用例执行失败时进行截图，详情请见本书9.3.2节中介绍的结合Spoon出错重试与截图。

除了常规的操作外，Robotium测试框架还提供了发送模拟按键`sendKeys(int key)`、设置屏幕是横屏还是竖屏`setActivityOrientation(int orientation)`、模拟点击返回键`goBack()`、跳转至指定Activity的方法`goBackToActivity(String name)`、收起输入法`hideSoftKeyboard()`、关闭所有已打开的Activity的方法`finishOpenedActivities()`等。

通过组合利用这些常用操作，基本就可以完成在Android端的UI自动化操作了。

3.WebView支持

在Android App中由于HTML可以更快地响应变化，而不像Native那样需要发布版本才能让用户使用上新特性，因此大多数App都是既有Native部分，也有HTML部分，也即俗称的Hybrid App。而Robotium在Robotium4.0版本中就开始全面支持WebView的自动化了。要了解如何使用Robotium测试框架来对App中的WebView部分进行自动化测试，首先需要了解HTML基础，然后了解Robotium是如何获取页面元素并进行操作的。

1) HTML基础

Robotium支持通过ID、className等方式来获取WebElement元素，因此，首先了解ID、className等的概念，模拟打开GitHub首页并查看网页源码如图3-12所示。

·HTML元素：指的是从开始标签到结束标签的所有代码。如图3-12所示，Sign in按钮在开始标签

·HTML属性：属性总是以名称/值对的形式出现的，比如：
name="value"。属性总是在HTML元素的开始标签中规定的。核心属性有class（规定元素的类名）、ID（规定元素的唯一ID）。Sign in按钮中就有class属性，class="btn btn-block primary"。

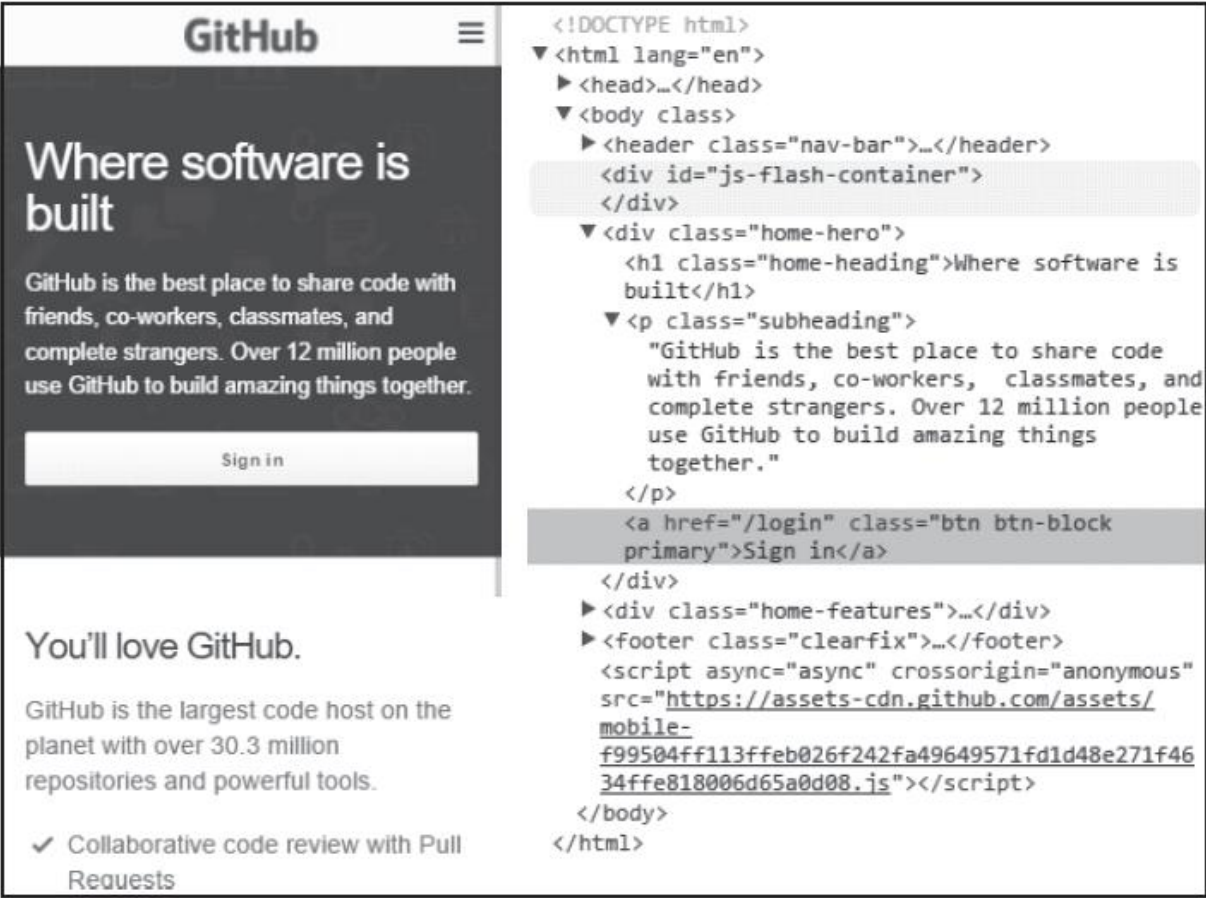


图3-12 GitHub首页的HTML结构

2) WebElement相关API及操作

WebElement相关API见表3-9。

表3-9 WebElement相关API

返回值	方法及说明
ArrayList<WebElement>	getCurrentWebElements() 获取当前 WebView 的所有 WebElement 元素
ArrayList<WebElement>	getCurrentWebElements(By by) 通过 By 根据指定的元素属性获取当前 WebView 的所有 WebElement 元素
void	clickOnWebElement(By by) 通过 By 根据指定的元素属性点击 WebElement
void	clickOnWebElement(WebElement webElement) 点击指定的 WebElement
void	enterTextInWebElement(By by, String text) 根据 by 找到 WebElement，并输入指定的文本 text
boolean	waitForWebElement(By by) 等待根据 by 获得的 WebElement 出现

在Robotium中对WebElement进行操作有两种方式，一种是先获取相应的WebElement，然后发送点击事件，另一种则是直接调用clickOnWebElement（By by）进行点击。

在获取WebElement元素前我们首先需要知道这个页面的HTML结构，需要知道URL链接才能方便地查看HTML元素、属性等。

获取WebView中的页面信息可以参考本书6.3.3节Appium脚本常见问题及处理方法中如何获取WebView中的页面信息这一部分内容，通过Chrome浏览器中的DevTools工具可以快速方便地查看WebView中的信息。

我们也可以使用原始的如代码清单3-3所示的方式打印出所有的元素信息。

代码清单3-3 使用日志打印方式获取元素信息


```
ArrayList<WebElement> webElements = solo.getCurrentWebElements();
WebElement webElement = null;
for(int i=0;i< webElements.size();i++){
    webElement = webElements.get(i);
    Log.i("WebElement", "getId:" + webElement.getId());
    Log.i("WebElement", "getClassName:"+webElement.getClassName());
    Log.i ("WebElement", "getText:" + webElement.getText());
}
```

当我们知道了相应页面的元素、属性后，就可以通过元素或属性等信息来获取指定的WebElement。

1) 获取当前WebView所有WebElement

```
ArrayList<WebElement> webElements = solo.getCurrentWebElements();
```

2) 通过className获取

```
ArrayList<WebElement> signIns = solo.getCurrentWebElements(By
    .className("btn btn-block primary"));
```

3) 通过ID获取

```
ArrayList<WebElement> signIns = solo.getCurrentWebElements(By
    .id("example_id"));
```

4) 通过textContent获取

```
ArrayList<WebElement> signIns = solo.getCurrentWebElements(By
    .textContent("Sign in"));
```

类似的还有通过cssSelector、name、tagName、xpath等方式获取。

5) 通过WebElement点击

拿到WebElement后，如果在页面中该标识是唯一的，那么数组长度为1，可以通过clickOnWebElement (WebElement webElement) 方法比较精确地点击。

```
solo.clickOnWebElement(signIns.get(0));
```

以上获取WebElement并点击也可以直接使用clickOnWebElement (By by) 方法完成。

```
solo.clickOnWebElement(By.className("btn btn-block primary"));
```

6) WebElement输入

```
solo.enterTextInWebElement(By.name("userId"), "your username");  
solo.enterTextInWebElement(By.name("passwd"), "your passwd");
```

同样地，WebElement也支持等待操作，可以通过waitForWebElement (By by) 等待相应的元素出现，然后查找，这样可以使脚本更健壮。不过Robotium中的clickOnTx-WebElement (By by) 也均默认已经使用了等待机制，因此非特殊情况，脚本中不需要额外增加等待操作。



注意

Robotium中对WebView的支持由于是使用对系统WebView执行JS从而封装获取页面元素的方式，因此该测试框架只支持App中使用系统WebView的情况，如果App或浏览器使用的是非系统内核的WebView，例如腾讯手机QQ浏览器的X5内核，则无法使用，需要引用X5的SDK并对Robotium进行改造才可支持。

4.断言

自动化测试中，我们获取控件、执行操作后，接下来就是要对操作后的场景进行断言了。Robotium是基于Instrumentation的测试框架，其测试用例编写的框架是基于Junit的，因此，本小节将先介绍Junit中的断言，然后介绍Robotium中适用于Android端自动化的断言。

1) Junit中的断言

Junit中的断言相关API见表3-10。

表3-10 Junit中的断言相关API

返回值	方法及说明
void	assertTrue(String message, boolean condition) 断言传入的 condition 参数应该为 True，否则将抛出一个带有 message 提示的 Throwable 异常
void	assertFalse(String message, boolean condition) 断言传入的 condition 参数应该为 False，否则将抛出一个带有 message 提示的 Throwable 异常
void	fail(String message) 直接使用例失败，并抛出一个带有 message 提示的 Throawable 异常

JUnit中的断言可以查看Android SDK中junit.framework.Assert包下的Assert类，常用的有assertTrue (String message, boolean condition) 方法，即断言方法中第二个参数condition的结果是否为True，如果为True则该语句执行通过，否则该语句将抛出Throwable的异常，而异常中的提示语将为第一个参数message。因此，使用断言时，应该准确明了地说明message参数，以便断言不符合预期时可以快速判断是什么原因导致的。例如断言某个控件应该要显示在界面中，代码如下：

```
Button loginBtn = (Button) solo.getView("loginBtn");
assertTrue("'登录'按钮应该要显示在界面", loginBtn.isShown());
```

同样地，还有assertFalse (String message, boolean condition) 方法，用于断言第二个条件中的结果应该为False。通过这两个方法，只要测试过程中的预期结果能转换成True或False的都可以进行判断，例如判断界面元素是否显示、数值大小比较、文本对比等。

在测试工程中，当出现某种场景时，有时我们希望直接使用例失败而不再往下执行，此时可以使用Assert类中的fail (String message) 方法，例如：

```
if(isBadHappened()){
    fail("this should no happened");
}
```

而如果出现某种场景，我们希望直接使用例通过而不再执行，则此时在用例脚本中直接使用return即可。

2) Robotium中的断言

Robotium中的断言相关API见表3-11。

表3-11 Robotium中的断言相关API

返回值	方法及说明
void	assertCurrentActivity(String message, String name) 断言当前界面是否为 name 参数指定的 Activity，若不是则抛出一个带有 message 提示的 Throwable 异常
void	assertMemoryNotLow() 断言当前是否处于低内存状态

Robotium基于Junit中的断言判断，也封装了几个方便在Android端自动化时使用的断言方法。例如assertCurrentActivity（String message，String name）方法可以判断当前界面是否是预期的Activity，我们知道Android中许多页面都对应于一个Activity，当App跳转到一个界面时，就可以使用该方法来判断是否已跳转到相应Activity了。

```
// 获取当前的
Activity名

String currentActivity =
    solo.getCurrentActivity().getClass().getSimpleName();
// expectedActivity为期望跳转的
Activity
solo.assertCurrentActivity("expected xxxActivity" + " but was " + currentActivity,
expectedActivity);
```

另外，测试框架中的assertMemoryNotLow（）方法可以用来判断当前是否处于内存吃紧的情况。在Robotium封装的断言API并不多，因为如前文所说，大多数场景都可以使用True或False来进行判断。

3) Android中的断言

Android中的断言相关API见表3-12。

表3-12 Android中的断言相关API

返回值	方法及说明
void	assertOnScreen(View origin, View view) 断言 view 是否在屏幕中
void	assertBottomAligned(View first, View second) 断言两个 view 是否底端对齐，即它们的底端 y 坐标相等

在Android SDK中， android.test.ViewAsserts包下有个ViewAsserts类可以方便地进行与控件相关的断言。例如断言控件是否在窗口中 assertOnScreen（View origin， View view），断言两个控件是否底部对齐assertBottomAligned（View first， View second），是否右对齐 assertRightAligned（View first， View second），等等。而之所以能实现这些断言在于View控件本身就具有非常多的可以用于判断自身状态的属性，例如View可以判断自身是否显示isShown（），判断是否被选中 isSelected（），还可以获取自身所在的坐标位置getLocationOnScreen（int[]location）和宽高getWidth（）、getHeight（），等等。由于基于Robotium编写的测试用例是以App形式安装进手机的，且运行时是运行

在被测应用所在的进程，因此我们使用断言时，可以借助Android SDK中丰富的类库来进行各种判断，例如判断当前网络状态、应用安装情况、当前应用是否处于前台等，可以很方便地对测试的预期结果进行判断。如代码清单3-4所示，调用Android中的API根据包名判断是否是系统应用。

代码清单3-4 根据包名判断是否是系统应用

```
/**
 * 根据
 * packageName判断该应用是否是系统应用
 *
 * @param packageName 应用的包名
 *
 * @return true, 系统应用;
 * false, 非系统应用
 */
public boolean isSystemApp(String packageName){
    PackageManager pm =
    getInstrumentation().getTargetContext().getApplicationContext().getPackageManager(
    );
    ApplicationInfo applicationInfo = null;
    try {
        applicationInfo = pm.getApplicationInfo(packageName,
        PackageManager.GET_UNINSTALLED_PACKAGES);
        if(applicationInfo !=null && (applicationInfo.flags &
        ApplicationInfo.FLAG_SYSTEM) ==1){
            LogUtils.logD(TAG, "applicationInfo flag:" + (applicationInfo.flags &
            ApplicationInfo.FLAG_SYSTEM));
            return true;
        }
    } catch (NameNotFoundException e) {
    }
    return false;
}
```

3.2 Robotium原理简析

如前文所述，一个基本的自动化测试用例主要分为获取控件、控件操作、断言三个步骤，而在实际编写测试用例的过程中，我们常常会遇到各种各样的问题，比如：

- 在这样的UI结构下该如何获取控件？
- 为何报这样或那样的错？
- 明明滑动了为何没有效果？

因为不同的项目有其自身的独特性与复杂性，没有任何书籍可以解决实际过程中遇到的所有问题，甚至即使求助Google搜索也可能得不到自己想要的答案。因此，对于任何一门技术而言都很有必要知其然并知其所以然，只有了解了其原理实现，才能更高效地运用在实际项目中。本节将从获取控件原理、控件操作原理、WebView支持等维度来对Robotium原理进行简要解析。

3.2.1 Robotium支持Native原理

1.获取控件原理

我们知道Android会为res目录下的所有资源分配ID，例如在布局xml文件中使用了`android: id="@+id/example_id"`，那么在Android工程编译时就会在R.java中相应地为该布局控件分配一个int型的ID，在Android工程中就可以通过Activity、Context或View等对象调用`findViewById (int id)`方法引用相应布局中的控件。因此，在测试工程中，如果是在源码的情况下，测试工程可以引用被测工程的代码，也即可以直接获得被测工程中R.java中的ID，因此可以通过这种方式直接根据ID获取控件。Robotium中根据ID获取控件的实现即包含该方式，如代码清单3-5所示。

代码清单3-5 Getter.getView

```
public View getView(int id, int index, int timeout){
    final Activity activity = activityUtils.getCurrentActivity(false);
    View viewToReturn = null;
    //如果
    index小于
    1. 则直接通过
    Activity的
    findViewById查找

    if(index < 1){
        index = 0;
```

```
        viewToReturn = activity.findViewById(id);
    }
    if (viewToReturn != null) {
        return viewToReturn;
    }
    return waiter.waitForView(id, index, timeout);
}
```

在`getView (int id, int index, int timeout)`方法中，先获取当前所在的`Activity`，然后直接通过`findViewById (id)`方法尝试获取控件，如果该方法能够正确获取，则直接返回；否则，使用`waitForView (id, index, timeout)`方法进一步等待控件的出现。

对于测试工程没有关联被测工程的情况，是无法直接通过`R.id.example_id`的形式获取控件的，此时一般调用`getView (String id)`方法，即通过`String`型ID获取。之所以可以通过`String`型ID获取控件，是因为`Robotium`中该方法使用了`Resources.getIdentifier (String name, String defType, String defPackage)`方法动态地将`String`型ID转换成了`int`型ID，如代码清单3-6所示。

代码清单3-6 Getter.getView (String id, int index)

```
public View getView(String id, int index){
    View viewToReturn = null;
    Context targetContext = instrumentation.getTargetContext();
    String packageName = targetContext.getPackageName();
    //先将

String类型的

ID转换成

int型的

ID
    int viewId = targetContext.getResources().getIdentifier(id, "id",
packageName);
}
```

```

        if(viewId != 0){
            viewToReturn = getView(viewId, index, TIMEOUT);
        }
        //如果还未找到, 则传入的
        ID可能是
        Android系统中的
        ID
        if(viewToReturn == null){
            int androidViewId = targetContext.getResources().getIdentifier(id, "id",
"android");
            if(androidViewId != 0){
                viewToReturn = getView(androidViewId, index, TIMEOUT);
            }
        }
        if(viewToReturn != null){
            return viewToReturn;
        }
        return getView(viewId, index);
    }

```

因此, 为了简化操作, 我们完全可以统一使用`getView (String id)`方法来获取控件。

以上为根据ID获取控件的一种方式, 另一种方式则是通过`WindowManager`获取所有View后再进行各种过滤封装。如代码清单3-7所示, 在`ViewFetcher`中通过`getAllViews`方法获取所有的View, 其中分别处理`DecorView`与`nonDecorView`。

代码清单3-7 ViewFetcher.getAllViews

```

public ArrayList<View> getAllViews(boolean onlySufficientlyVisible) {
    //获取所有的
    DecorViews
    final View[] views = getWindowDecorViews();
    final ArrayList<View> allViews = new ArrayList<View>();
    final View[] nonDecorViews = getNonDecorViews(views);
    View view = null;
    if(nonDecorViews != null){
        for(int i = 0; i < nonDecorViews.length; i++){
            view = nonDecorViews[i];
        }
    }
    allViews.addAll(views);
    if(onlySufficientlyVisible){
        allViews.removeIf(view -> !view.isShown());
    }
    return allViews;
}

```

```

        try {
            addChildren(allViews, (ViewGroup)view, onlySufficientlyVisible);
        } catch (Exception ignored) {}
        if(view != null) allViews.add(view);
    }
}
if (views != null && views.length > 0) {
    view = getRecentDecorView(views);
    try {
        addChildren(allViews, (ViewGroup)view, onlySufficientlyVisible);
    } catch (Exception ignored) {}
    if(view != null) allViews.add(view);
}
return allViews;
}

```

如代码清单3-8所示，在getWindowDecorViews方法中通过使用反射获取Window-Manager中的mViews对象来获取所有DecorView，其中也可以看到对于Android系统版本大于19的处理是不同的。

代码清单3-8 ViewFetcher.getWindowDecorViews

```

@SuppressWarnings("unchecked")
public View[] getWindowDecorViews()
{
    Field viewsField;
    Field instanceField;
    try {
        //通过反射获取
        WindowManagerGlobal或
        WindowManagerImpl中的
        mViews变量

        viewsField = windowManager.getDeclaredField("mViews");
        //通过反射获取
        WindowManagerGlobal或
        WindowManagerImpl中的
        WindowManager实例的变量

        instanceField = windowManager.getDeclaredField(windowManagerString);
        viewsField.setAccessible(true);
        instanceField.setAccessible(true);
    }
}

```

```
        Object instance = instanceField.get(null);
        View[] result;
        if (android.os.Build.VERSION.SDK_INT >= 19) {
            result = ((ArrayList<View>) viewsField.get(instance)).toArray(new
View[0]);
        } else {
            result = (View[]) viewsField.get(instance);
        }
        return result;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

再看代码清单3-9中的WindowManagerString变量的来源，如代码清单3-9所示，WindowManagerString也同样地需要根据Android系统版本的不同而分别处理。

代码清单3-9 ViewFetcher.setWindowManagerString

```
private void setWindowManagerString(){
    //不同的系统版本,
    WindowManager的变量名不同

    if (android.os.Build.VERSION.SDK_INT >= 17) {
        windowManagerString = "sDefaultWindowManager";
    } else if (android.os.Build.VERSION.SDK_INT >= 13) {
        windowManagerString = "sWindowManager";
    } else {
        windowManagerString = "mWindowManager";
    }
}
```

至此我们知道了Robotium中获取所有Views是通过反射机制实现的，而源码中的变量很可能根据版本的不同而改变，因此通过反射则往往需根据系统版本的不同而分别处理。所以，使用Robotium时最好

使用开源项目中的最新版本，因为当有新的Android系统版本发布时，很可能Robotium也需要与时俱进地完善获取控件方式。

2.控件操作原理

Robotium获取控件后，调用clickOnView（View view）方法就可以完成点击操作，这个方法可以实现两大功能：

- 根据View获取了控件在屏幕中的坐标。
- 根据坐标发送了模拟的点击操作。

如代码清单3-10所示，由于View本身可以获取到在屏幕中的起始坐标与控件长宽，因此通过getLocationOnScreen获取起始坐标后，再加上1/2的长与宽，即可计算出控件的中心点在屏幕中的位置。

代码清单3-10 Clicker.getClickCoordinates

```
private float[] getClickCoordinates(View view){
    sleeper.sleep(200);
    int[] xyLocation = new int[2];
    float[] xyToClick = new float[2];
    //获取
    view的坐标,
    xyLocation[0]为
    x坐标的值,
    xyLocation[1]为
    y坐标的值

    view.getLocationOnScreen(xyLocation);
```

```
final int viewWidth = view.getWidth();
final int viewHeight = view.getHeight();
//xyLocation中的值为控件左上角的坐标，因此

xyLocation[0]+宽长除

2即为该控件在

x轴的中

心点，同样地计算在

y轴的中心点

final float x = xyLocation[0] + (viewWidth / 2.0f);
float y = xyLocation[1] + (viewHeight / 2.0f);
xyToClick[0] = x;
xyToClick[1] = y;
return xyToClick;
}
```

知道了需要点击的位置后，那么接下来发送模拟点击就可以了。Android中的模拟操作可以通过MotionEvent来实现，而MotionEvent主要有以下三种形式：

- MotionEvent.ACTION_DOWN：模拟对屏幕发送下按事件。
- MotionEvent.ACTION_UP：模拟对屏幕发送上抬事件。
- MotionEvent.ACTION_MOVE：模拟对屏幕发送移动事件。

Robotium中的点击屏幕方法即是通过MotionEvent实现的，如代码清单3-11所示，通过MotionEvent.obtain（long downTime，long eventTime，int action，float x，float y，int metaState）方法获取相应的event事件后，再通过Instrumentation的sendPointerSync（MotionEvent event）方法将event事件实际地在手机上模拟执行。

代码清单3-11 Clicker.clickOnScreen

```
public void clickOnScreen(float x, float y, View view) {
    boolean successfull = false;
    int retry = 0;
    SecurityException ex = null;
    while(!successfull && retry < 20) {
        long downTime = SystemClock.uptimeMillis();
        long eventTime = SystemClock.uptimeMillis();
        MotionEvent event = MotionEvent.obtain(downTime, eventTime,
            MotionEvent.ACTION_DOWN, x, y, 0);
        MotionEvent event2 = MotionEvent.obtain(downTime, eventTime,
            MotionEvent.ACTION_UP, x, y, 0);
        try{
            //通过
            Instrumentation模拟发送下按操作

            inst.sendPointerSync(event);
            //通过

            Instrumentation模拟发送上抬操作，与下按操作结合，模拟完成了一个点击过程

            inst.sendPointerSync(event2);
            successfull = true;
        }catch(SecurityException e){
            ex = e;
            dialogUtils.hideSoftKeyboard(null, false, true);
            sleeper.sleep(MINI_WAIT);
            retry++;
            View identicalView = viewFetcher.getIdenticalView(view);
            if(identicalView != null){
                float[] xyToClick = getClickCoordinates(identicalView);
                x = xyToClick[0];
                y = xyToClick[1];
            }
        }
    }
    //如果点击失败，将抛出异常

    if(!successfull) {
        Assert.fail("Click at (" + x + ", " + y + ") can not be completed! (" + (ex != null
        ? ex.getClass().getName() + ": " + ex.getMessage() : "null") + ")");
    }
}
```

结合getClickCoordinates（View view）与clickOnScreen（float x, float y, View view）方法就完成了clickOnView（View view）方法的核

心实现。通过控制不同手势操作的时间顺序还可以模拟各种手势操作，例如先发送`MotionEvent.ACTION_DOWN`，一段时间后，再发送`MotionEvent.ACTION_UP`就模拟了长按操作。先发送`MotionEvent.ACTION_DOWN`，然后发送`MotionEvent.ACTION_MOVE`，最后发送`MotionEvent.ACTION_UP`就是滑动操作了。因此，结合`MotionEvent`的各种模拟事件也可以自行实现自定义的手势操作。

3.2.2 Robotium支持WebView原理

在上一节中我们介绍了在Robotium中如何通过By.id或By.className方式获取Web-Element，那么Robotium中是如何获取到相应的HTML元素，并能知道元素坐标，从而发送点击事件的呢？

1.WebElement对象

Robotium中以WebElement对象对HTML元素进行了封装，在这个WebElement对象中包含locationX、locationY、ID、text、name、className、tagName等信息。

·locationX、locationY：标识该HTML元素在屏幕中所在的X坐标和Y坐标。

·ID、className：该HTML元素的属性。

·tagName：该HTML元素的标签。

Robotium中封装了WebElement，提供了clickOnWebElement (WebElement webElement) ，
ArrayList<WebElement>getCurrentWebElements () 等操作Web元素的

API，对于在Android客户端中展示的Web页面，Robotium是如何把里面的元素都提取出来，并封装进WebElement对象中的呢？

如图3-13所示，通过getWebElements方法的调用关系图可以看出，Robotium主要通过JS注入的方式获取Web页面所有的元素，再对这些元素进行提取并封装成WebElement对象。在Android端与JS交互则离不开WebView和WebCromeClient。

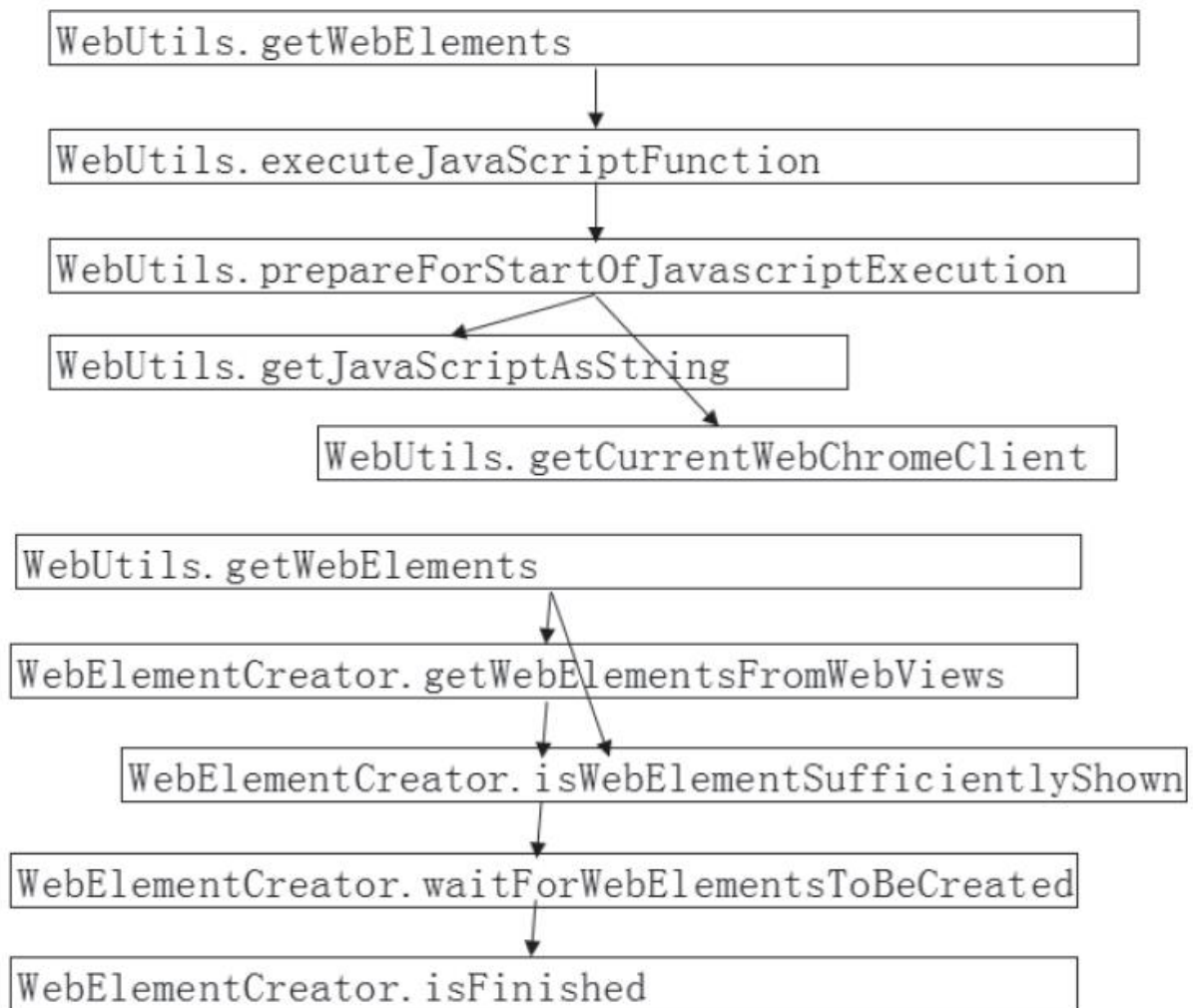


图3-13 getWebElements方法的调用关系图

2.WebElement元素获取

1) 利用JS获取页面中的所有元素

在PC上，获取网页的元素可以通过注入javascript元素来完成，以Chrome浏览器为例，打开工具——JavaScript控制台（快捷方式：Ctrl+Shift+J键），输入javascript: prompt (document.URL) 即会弹出含当前页面的URL的提示框，因此通过编写适当的JS脚本就可以在这个弹出框中显示所有的页面元素。RobotiumWeb.js就提供了获取所有HTML元素的JS脚本。以Solo中getWebElements () 为例，如代码清单3-12所示，可分为两步，先通过executeJavaScriptFunction () 方法执行JS脚本，然后根据执行结果通过getWebElements返回。

代码清单3-12 WebUtils.getWebElements

```
public ArrayList<WebElement> getWebElements(boolean onlySufficientlyVisible){
    boolean javaScriptWasExecuted =
    executeJavaScriptFunction("allWebElements()");
    return getWebElements(javaScriptWasExecuted, onlySufficientlyVisible);
}
```

如代码清单3-13所示，在executeJavaScriptFunction (final String function) 方法中通过webView.loadUrl (String url) 方法执行JS，而这里的WebView是通过getCurrentViews (Class<T>classToFilterBy, boolean includeSubclasses) 过滤出来的，且是过滤的

android.webkit.WebView，这也是Robotium只支持系统WebView而不支持第三方浏览内核中的WebView的原因：

代码清单3-13 WebUtils.executeJavaScriptFunction

```
private boolean executeJavaScriptFunction(final String function) {
    List<WebView> webViews = viewFetcher.getCurrentViews(WebView.class, true);
    //获取当前屏幕中最新的

    WebView, 即目标要执行

    JS的

    WebView
    //注: 这里获取的

    WebView可能不是目标

    WebView, 那么将导致获取

    WebElement失败

    final WebView webView = viewFetcher.getFreshestView((ArrayList<WebView>)
webViews);
    if(webView == null) {
        return false;
    }
    //执行

    JS前的准备工作, 如设置

    WebSettings、获取

    JS方法等

    final String javaScript =
setWebFrame(prepareForStartOfJavascriptExecution(webViews));
    inst.runOnMainSync(new Runnable() {
        public void run() {
            if(webView != null){
                //调用

                loadUrl执行

                JS

                webView.loadUrl("javascript:" + javaScript + function);
            }
        }
    });
    return true;
}
```

想返回什么样的结果，关键在于执行了什么样的JS方法，Robotium中的getWeb-Elements（）执行的JS方法是allWebElements（），代码片段可以通过RobotiumWeb.js找到，如代码清单3-14所示，采用遍历DOM的形式获取所有的元素信息。

代码清单3-14 RobotiumWeb.js中的allWebElements（）

```
function allWebElements() {
    for (var key in document.all){
        try{
            promptElement(document.all[key]);
        }catch(ignored){}
    }
    finished();
}
```

如代码清单3-15所示，将代码清单3-15中遍历获取到的每一个元素分别获取ID、text、className等，然后将元素通过prompt方法以提示框形式显示。在prompt时，会在ID、text、className等字段之间加上';'，'特殊字符，以便解析时区分这几个字段。

代码清单3-15 RobotiumWeb.js中的promptElement（element）

```
function promptElement(element) {
    var id = element.id;
    var text = element.innerText;
    if(text.trim().length == 0){
        text = element.value;
    }
    var name = element.getAttribute('name');
    var className = element.className;
    var tagName = element.tagName;
    var attributes = "";
    var htmlAttributes = element.attributes;
    for (var i = 0, htmlAttribute; htmlAttribute = htmlAttributes[i]; i++){
        attributes += htmlAttribute.name + ":@" + htmlAttribute.value;
        if (i + 1 < htmlAttributes.length) {
            attributes += "#$";
        }
    }
}
```

```

    }
  }
  var rect = element.getBoundingClientRect();
  if(rect.width > 0 && rect.height > 0 && rect.left >= 0 && rect.top >= 0){
    prompt(id + ';;' + text + ';;' + name + ";;" + className + ";;" + tagName
+ ";;" + rect.left + ';;' + rect.top + ';;' + rect.width + ';;' + rect.height +
';;' + attributes);
  }
}

```

最后，执行finished（）方法，调用prompt提示框，提示语为特定的'robotium-finished'，用于在Robotium执行JS时，判断是否执行完毕，如代码清单3-16所示。

代码清单3-16 RobotiumWeb.js中的finished（）

```

function finished(){
    //robotium-finished用来标识

    Web元素遍历结束

    prompt('robotium-finished');
}

```

通过JS完成了Web页面所有元素的提取，提取的所有元素是以prompt方式显示在提示框中的，那么提示框中包含的内容在Android中怎么获取呢？

2) 通过onJsPrompt回调获取prompt提示框中的信息

如代码清单3-17所示，通过JS注入获取到Web页面所有的元素后，可以通过onJsPrompt回调来对这些元素进行提取。Robotium写了个继承自WebChromeClient类的RobotiumWebClient类，覆写了onJsPrompt用于

回调提取元素信息，如果提示框中包含“robotium-finished”字符串，即表示这段JS脚本执行完毕了，此时通知webElementCreator可以停止等待，否则，将不断将prompt框中的信息交由webElementCreator.createWeb-ElementAndAddInList解析处理。

代码清单3-17 RobotiumWebClient中的onJsPrompt

```
@Override
public boolean onJsPrompt(WebView view, String url, String message, String
    defaultValue, JsPromptResult r) {
    //当
    message包含
    robotium-finished时, 表示
    JS执行结束

    if(message != null && (message.contains(";") || message.contains("robotium-
    finished"))){
        if(message.equals("robotium-finished")){
            //setFinished为
            true后,
            WebElementCreator将停止等待

            webElementCreator.setFinished(true);
        }
        else{
            webElementCreator.createWebElementAndAddInList(message, view);
        }
        r.confirm();
        return true;
    }
    else {
        if(originalWebChromeClient != null) {
            return originalWebChromeClient.onJsPrompt(view, url, message,
            defaultValue, r);
        }
        return true;
    }
}
```

3) 将回调中获取的元素信息封装进WebElement对象中

获取到onJsPrompt回调中的元素信息后，接下来就可以对这些已经过处理、含特殊格式的消息进行解析了，依次得到WebElement的ID、text、name等字段。如代码清单3-18所示，将information通过特殊字符串“; , ”分隔成数组对该字符串进行分段解析，将解析而得的ID、text、name及x、y坐标存储至WebElement对象中。

代码清单3-18 WebElementCreator中的 createWebElementAndSetLocation

```
private WebElement createWebElementAndSetLocation(String information, WebView
webView){
    //将
    information通过特殊字符串“
    ; , ”分隔成数组

    String[] data = information.split("; , ");
    String[] elements = null;
    int x = 0;
    int y = 0;
    int width = 0;
    int height = 0;
    Hashtable<String, String> attributes = new Hashtable<String, String>();
    try{
        x = Math.round(Float.valueOf(data[5]));
        y = Math.round(Float.valueOf(data[6]));
        width = Math.round(Float.valueOf(data[7]));
        height = Math.round(Float.valueOf(data[8]));
        elements = data[9].split("\\#\\$");
    }catch(Exception ignored){}
    if(elements != null) {
        for (int index = 0; index < elements.length; index++){
            String[] element = elements[index].split("::");
            if (element.length > 1) {
                attributes.put(element[0], element[1]);
            } else {
                attributes.put(element[0], element[0]);
            }
        }
    }
    WebElement webElement = null;
    try{
        //设置
        WebElement中的各个字段
```

```
        webElement = new WebElement(data[0], data[1], data[2], data[3], data[4],
attributes);
        setLocation(webElement, webView, x, y, width, height);
    }catch(Exception ignored) {}
    return webElement;
}
```

这样，把JS执行时提取到的所有元素信息解析出来，并储存至WebElement对象中，在获取到相应的WebElement对象后，就包括了元素的ID、text、className等属性及其在屏幕中的坐标，完成了对Web自动化的支持。

3.3 Robotium实践运用

3.3.1 控件ID相同时获取控件

实际界面中常常有一些子控件是相同ID甚至没有ID的，但这时候一般其父视图是有ID的。如图3-14所示，每个TAB的控件ID是相同的。



图3-14 拥有相同ID的底部TAB

因为界面中也很有可能会出现多个发现、游戏这样的文本，因此也不能采取类似getText（“发现”）这样的方式。这里，我们就可以通过ID获取唯一父控件，再通过过滤方式获取指定的控件。

```
//先根据
ID获得唯一的布局
LinearLayout
LinearLayout mTabs = (LinearLayout)solo.getView("main_tabs");
//然后通过过滤方式获取该
LinearLayout下的所有文本控件

ArrayList<TextView> tabs = solo
    .getCurrentViews(TextView.class,mTabs);
```

如果子控件的ID都是一样的，而我们仍然希望通过ID来定位控件，那么应该如何获取呢？我们知道不论是Activity类还是View类都是可以通过findViewById（int id）方法直接在控件树中根据ID来查找控件的，因此当我们获得一个父视图后，就可以通过findViewById（int id）方法根据ID来查找相应的子控件，这种方法可以普遍应用在ListView中。

```
//先根据
ID获得唯一的布局

ListView
ListView mListView = (ListView)solo.getView("example_list_id");
//先通过

mListView.getChildAt(0)获取该
ListView的第一个
child，然后再通过该

//child在控件树中使用

findViewById根据
ID来获取

TextView firstListTitle = (TextView) mListView.getChildAt(0).findViewById(getId
("example_title"));
```

这里的重点是findViewById（int id）传进去的是int型的ID，而我们通过hierarchyviewer或uiautomatorviewer查看到的ID都是String型的，由前文的原理介绍可知，我们可以将String型的ID转换成int型的ID，如代码清单3-19所示：

代码清单3-19 将String型的ID转换成int型的ID

```
public int getId(String id,String packageName){
    Context targetContext =
instrumentation.getTargetContext().getApplicationContext();
    int viewId = targetContext.getResources().getIdentifier(id, "id",
packageName);
    LogUtils.logD("CopyOfAssistantTabActivityTest", "viewId:" + viewId);
    if(viewId == 0){
        viewId = targetContext.getResources().getIdentifier(id, "id", "android");
    }
    return viewId;
}
```

因此，当碰到同一层级控件**ID**相同时，可以先寻找唯一的父布局，再通过父布局寻找子控件。如果子控件结构均相同，那么可以通过**index**索引来查找；如果子控件结构不一致，则可以通过遍历的方式找到指定的子控件。

3.3.2 ListView列表遍历

编写Android端的自动化测试用例，最常见的控件有ListView，而要想测试ListView，就必然要涉及ListView的遍历。

关于ListView的遍历，可能首先想到的是类似如代码清单3-20的实现方式。

代码清单3-20 设想中的列表遍历

```
for(int i=0;i<listView. getCount();i++){  
    listView.getChildAt(int index);  
    .....  
};  
}
```

但是，在Android中，对于listView.getChildAt（int index）而言，如果子控件是在屏幕之外的话，那么是无法点击的，因此要想点击或测试屏幕之外的子控件，就需要不断向上滑动。因此我们可以先遍历当前屏幕内的子控件，然后翻一屏，再遍历屏幕内的子控件，如此反复就可以遍历ListView所有的子控件了。

对于ListView而言，通过getFirstVisiblePosition（）和getLastVisiblePosition（）可以获取ListView在屏幕中第一个可见子控件及最后一个可见子控件在列表中的位置。当遍历至当前最后一个子

控件时，通过solo.scrollToLine (listView, lastPosition) 方法将列表滑至lastPosition所在的位置，即实现翻页的效果。当遍历至每个child子控件时，可以通过该子控件的布局结构来判断该子控件是否是要查找的控件。另外，需要注意的是，正如前文所介绍的，scrollToLine (listView, lastPosition) 方法并不会直接产生上滑手势，因此如果列表需要产生上滑动作才能加载更多他的话，则还需要配合使用drag方法进行上拉加载更多。

如代码清单3-21所示，遍历列表，查找列表中子节点为RelativeLayout且子节点的标题为×××的子控件。

代码清单3-21 遍历列表并找到指定标题的child

```
public RelativeLayout findCardByType(int maxCount) {  
    // 获取当前界面中的  
  
    ListView  
    ListView listView = getCurrentListView();  
    int firstPosition = 0;  
    int lastPosition = 0;  
    RelativeLayout relativeLayout = null;  
    int currentPosition = 1;  
    labelAll:  
    for (int i = 0; i < length; i++) {  
        firstPosition = listView.getFirstVisiblePosition();  
        lastPosition = listView.getLastVisiblePosition();  
        for (int j = 1; j <= lastPosition - firstPosition; j++) {  
            currentPosition++;  
            if (currentPosition >= maxCount) {  
                break labelAll;  
            }  
            // 判断该节点是否为  
  
        relativeLayout  
        if (listView.getChildAt(j) instanceof RelativeLayout) {  
            relativeLayout = (RelativeLayout) listView.getChildAt(j);  
            // 这里可以对该  
  
        relativeLayout进行判断，例如获取该
```

//RelativeLayout中的子控件，如果有标题则判断标题等

```
        if (isSatisfied(relativeLayout)) {  
            break labelAll;  
        }  
        relativeLayout = null;  
    }  
}  
solo.scrollToLine(listView, lastPosition);  
if (lastPosition >= listView.getCount()) {  
    // 当需要上拉加载更多时，调用
```

drag实现的方法进行上拉加载更多

```
        dragUpToShowAll(listView);  
    }  
    sleeper.sleep();  
}  
sleeper.sleep();  
return relativeLayout;  
}
```

3.3.3 修改Robotium以支持X5WebView

本节中的X5WebView指QQ浏览器团队出品的腾讯X5内核中的WebView。除了QQ、微信、应用宝等众多腾讯内部产品在使用X5内核外，京东、58同城等众多腾讯外部的合作伙伴也在使用X5内核。

腾讯X5网站：<http://x5.tencent.com/>。

然而Robotium本身并不支持获取X5WebView中的元素，因此无法对使用了X5内核的Web页面进行自动化测试，而通过3.2.2节中介绍的Robotium支持WebView原理可知，只要对Robotium稍加改造，即可使用同样的原理获取WebElement对象，完成对X5WebView自动化的支持。

这里再概述一下Robotium支持WebView的过程，以便理解为何Robotium不支持X5以及如何修改。

步骤1：获取目标WebView。

如代码清单3-13所示，代码final WebView
webView=viewFetcher.getFreshestView（viewFetcher.getCurrentViews
（WebView.class））；调用ViewFetcher类获取当前界面中的
WebView，而该WebView是android.webkit.WebView。

步骤2: 做执行JS前的准备工作。

如代码清单3-13所示, final String

javaScript=prepareForStartOfJavascriptExecution (); 调用
prepareForStartOfJavascriptExecution (), 该方法还调用了如代码清单
3-22所示的代码, 将WebSettings是否允许执行JS设置为True (系统默认
是False)。而且还设置了WebView的WebChromeClient
(WebChromeClient用于辅助WebView处理Javascript的对话框、提示框
等)。从这里可以看出Robotium使用的是继承自
android.webkit.WebChromeClient的RobotiumWebClient。

代码清单3-22

RobotiumWebClient.enableJavascriptAndSetRobotiumWebClientd

```
/**
 * Enables JavaScript in the given {@code WebViews} objects.
 *
 * @param webViews the {@code WebView} objects to enable JavaScript in
 */
public void enableJavascriptAndSetRobotiumWebClient(List<WebView> webViews,
WebChromeClient originalWebChromeClient){
    this.originalWebChromeClient = originalWebChromeClient;
    for(final WebView webView : webViews){
        if(webView != null){
            inst.runOnMainSync(new Runnable() {
                public void run() {
                    //WebSettings开启
JS
                    webView.getSettings().setJavaScriptEnabled(true);
                    webView.setWebChromeClient(robotiumWebClient);
                }
            });
        }
    }
}
```

步骤3: 在指定WebView中执行相应JS。

如代码清单3-13所示，最后调用webView.loadUrl
("javascript: "+javaScript+function)；方法在指定的WebView中执行
相应片段的JS代码。

从以上核心步骤中可以看出，Robotium不支持X5的原因在于，首先，其获取目录WebView时，是获取android.webkit.WebView中的WebView；其次，辅助处理JS的WebChromeClient也是继承自android.webkit.WebChromeClient。而X5内核中的WebView并不是继承自android.webkit.WebView，X5内核中的WebChromeClient也不是继承自android.webkit.WebChromeClient，因此Robotium没法获取X5内核中的目标WebView，也就没法在目标WebView中执行JS并提取WebElement元素。了解个中缘由后，就可以稍加改造以支持X5WebView。

如图3-15所示为以外部引用（即该jar包的类并不实际打包进测试工程，仅在IDE调试时用。当调用相应的类时，寻找的是被测工程中的相应的类）的方式导入X5提供的SDK。

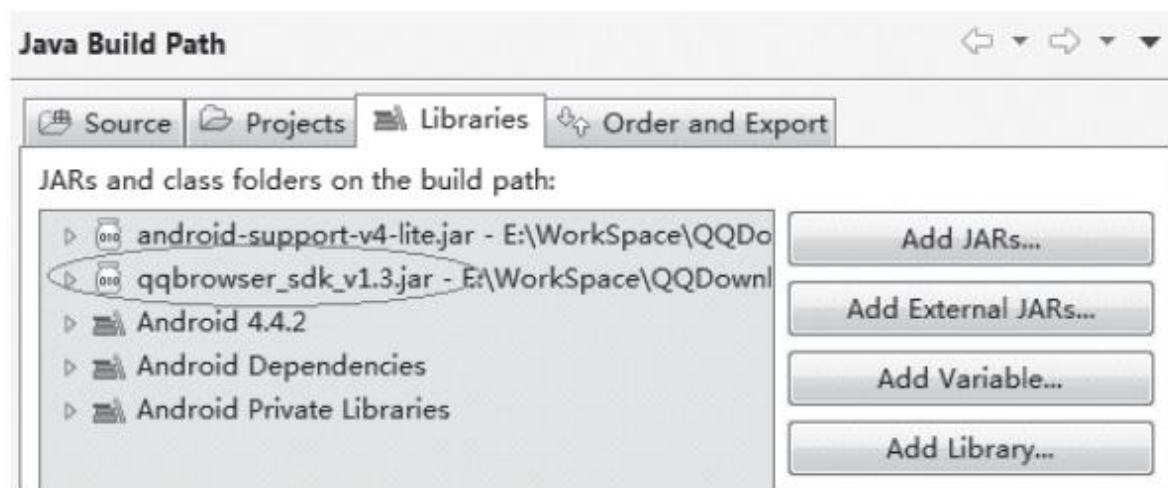


图3-15 导入X5提供的SDK

在获取目标WebView时，相应地修改成X5 SDK中的WebView。如图3-16所示，获取目标WebView时修改为
com.tencent.smtt.sdk.WebView。

```
/**
 * Executes the given JavaScript function
 *
 * @param function the function as a String
 * @return true if JavaScript function was executed
 */
private boolean executeJavaScriptFunction(final String function){
    final com.tencent.smtt.sdk.WebView webView = viewFetcher.
        getFreshestView(viewFetcher.getCurrentViews(com.tencent.smtt.sdk.WebView.class));

    if(webView == null){
        return false;
    }

    final String javaScript = prepareForStartOfJavascriptExecution();

    activityUtils.getCurrentActivity(false).runOnUiThread(new Runnable() {
        public void run() {
            if(webView != null){
                webView.loadUrl("javascript:" + javaScript + function);
            }
        }
    });
    return true;
}
```

图3-16 修改目标WebView

同样地，修改WebChromeClient为继承自com.tencent.smtt.sdk.WebChromeClient中的TxWebChromeClient，然后在WebView中设置WebChromeClient时使用TxWebChromeClient，如图3-17所示。

```
/**
 * Enables JavaScript in the given {@code WebView} objects.
 *
 * @param webViews the {@code WebView} objects to enable JavaScript in
 */
public void enableJavascriptAndSetRobotiumWebClient(List<com.tencent.smtt.sdk.WebView> webViews,
    com.tencent.smtt.sdk.WebChromeClient originalWebChromeClient){
    this.originalWebChromeClient = originalWebChromeClient;

    for(final com.tencent.smtt.sdk.WebView webView : webViews){

        if(webView != null){
            inst.runOnMainSync(new Runnable() {
                public void run() {
                    webView.getSettings().setJavaScriptEnabled(true);
                    webView.setWebChromeClient(robotiumWebClient);
                }
            });
        }
    }
}
```

图3-17 修改目标WebChromeClient

对于其他有相应的WebView或WebChromeClient调用的地方，均修改成X5 SDK中对应的WebView及WebChromeClient，修改完成后，将相应的类带上前缀以便区分，如图3-18所示。

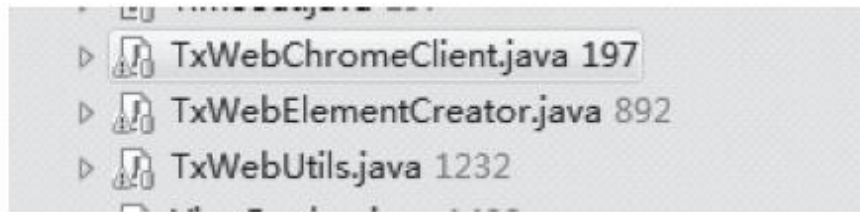


图3-18 修改后的类

当需要获取使用了X5内核的Web元素时，调用TxWebUtils类中的相应方法即可。如图3-19所示，与Robotium原有的WebUtils使用方法一致，至此，完成了对X5内核的支持。

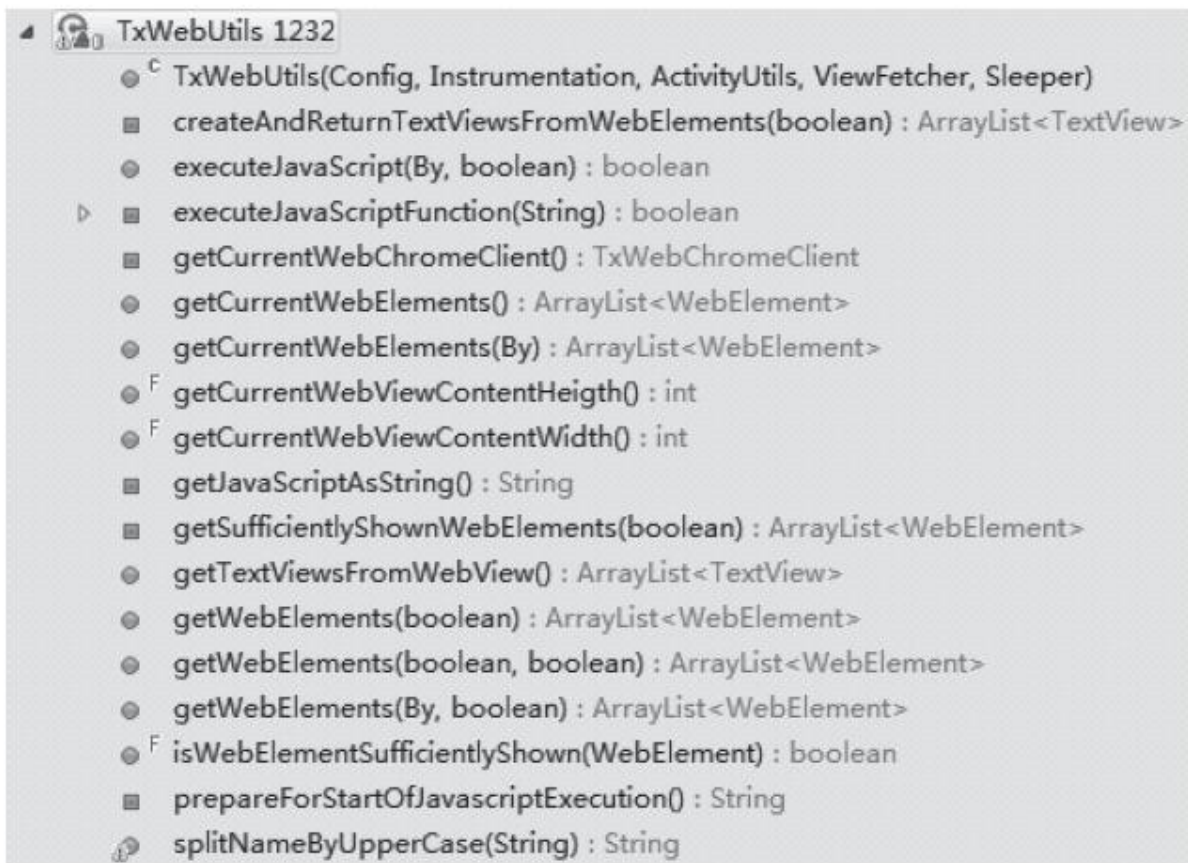


图3-19 TxWebUtils中的类方法

3.4 本章小结

本章分三小节，从功能、原理及实践三方面介绍了Robotium测试框架，第一小节先全面概览似的介绍了Robotium的整体，然后从控件获取、控件操作、WebView支持、断言等维度介绍了相应功能及其使用方法，力图让读者知道如何使用Robotium测试框架来进行用例编写。第二小节则分别从Native和Web角度介绍了Robotium的实现原理，力图让读者了解更多的为什么，从而可以在实际项目中更灵活地使用Robotium编写测试用例。第三小节则从实践运用角度选取一般项目中常见的一些场景，介绍使用Robotium处理的思路与方法。

本章主要从测试用例编写过程这一思维主线来介绍Robotium测试框架，从中也可以看出，不论是对获取复杂控件还是进行各类模拟操作，Robotium均可以很好地支持，且也可以支持App中的Web自动化，基本可以满足日常的自动化测试需求。当然，Robotium也有如跨应用能力弱等固有劣势，实际上也并没有哪一款测试框架可以解决所有遇到的测试问题，我们大可结合不同的测试框架、测试工具来解决实际问题。因此，Robotium可以说是一款不可多得的优秀的自动化测试框架。

第4章 Monkey基本原理及扩展应用

Monkey是Android系统自带的一款稳定性测试小工具，它以简单易用、方便快捷而广受测试者欢迎。本章分为四部分，由浅入深地为读者详细介绍Monkey工具。第一部分介绍Monkey基础知识，包括Monkey概况、常用参数、11大事件、环境搭建，以及Monkey命令行使用方法；第二部分介绍Monkey测试的基本方法，包括常规的稳定性测试、自定义脚本的稳定性测试、结合常用辅助命令的Monkey测试，以及Monkey日志的分析方法；第三部分介绍Monkey的原理，包括代码框架和代码逻辑分析；第四部分是Monkey使用的进阶篇，通过截图改造和Wi-Fi监控改造两个实际案例，介绍了如何通过修改Monkey源码达到优化Monkey工具的目的。本章结构知识图如图4-1所示。



图4-1 本章知识结构图

接下来，让我们开始来学习吧。

4.1 Monkey基础知识

4.1.1 Monkey概况

在Android的官方自动化测试领域有一只非常著名的“猴子”叫Monkey，这只“猴子”一旦启动，就会让被测的Android应用程序像猴子一样活蹦乱跳，到处乱跑。人们常用这只“猴子”来对被测程序进行压力测试，检查和评估被测程序的稳定性。

Android官方对这只“猴子”的描述是这样的：Monkey是Google提供的一个命令行工具，可运行在模拟器或实际设备中。它向系统发送伪随机的用户事件，模拟用户的按键输入、触摸屏输入、手势输入等，从而对正在运行的应用程序进行压力测试，目的是看设备多长时间会出现异常，并观察系统的稳定性和容错性能。

Monkey程序是Android系统自带的，其启动脚本是位于Android系统的/system/bin目录的Monkey文件，其jar包是位于Android系统的/system/framework目录的Monkey.jar文件。用户主要是通过adb命令来启动Monkey的，Monkey在运行时，会根据命令行参数的配置，生成伪随机的事件流，并在Android设备上执行对应的测试事件。同时，

Monkey还会对测试系统进行监测，当出现以下三种情况时会进行特殊处理：

- 如限定了**Monkey**运行在特定包上，当监测到试图转到其他包的操作，将对其进行阻止。

- 如应用程序崩溃或接收到任何失控异常，**Monkey**将记录对应的错误日志，并根据命令行参数判断是停止运行还是继续运行。

- 如果应用程序发生了程序无响应（**application not responding**）的错误，**Monkey**将记录对应的错误日志，并根据命令行参数判断是停止运行还是继续运行。

按照选定的不同级别的反馈信息，在**Monkey**中还可以看到其执行过程报告和生成的事件。

4.1.2 Monkey参数

Monkey启动的命令行脚本为：

```
monkey [options] <count>
```

其中，**options**表示Monkey执行的可配置参数，是可选项（如果不指定**options**，Monkey将以无反馈模式启动，并把事件任意发送到安装在目标环境中的全部包）；**count**表示Monkey执行的事件数，为必选项。

Options可简单划分为五类：

- 基本配置类参数。
- 事件类型和频率参数。
- 约束限制类参数。
- 调试类参数。
- 官方隐藏类参数。

以下是针对以上五种类型参数的详细介绍。

1.基本配置类参数

Monkey的基本配置类参数包括帮助参数和日志信息参数。帮助参数用于输出Monkey命令使用指导；日志信息参数将日志分为三个级别，级别越高，日志的信息越详细。具体参数信息见表4-1。

表4-1 Monkey基本配置类参数表

参数	说明
--help	输出 Monkey 的命令行使用方法
-v	表示反馈信息的级别，Monkey 命令行中每增加一个 -v 参数，Monkey 日志反馈信息的级别会对应增加一个 Level。Level 0（缺省值）除启动提示、测试完成和最终结果之外，提供较少信息。Level 1（-v-v）提供较为详细的测试信息，如逐个发送到 Activity 的事件。Level 2（-v-v-v）提供更加详细的设置信息，如测试中被选中的或未被选中的 Activity 等 举例：adb shell monkey -v-v 10

2.事件类型和频率参数

Monkey的事件类参数的作用是对随机事件进行调控，从而使其遵照设定运行，如设置各种事件的百分比、设置事件生成所使用的种子值等。频率参数主要限制事件执行的时间间隔。这两类的详细参数介绍见表4-2。

3.约束限制类参数

Monkey的约束限制类参数的作用是将随机事件运行的范围限制在一个或多个包或类中。详细参数介绍见表4-3。

表4-2 Monkey事件类型和频率参数表

参数	说明
<code>-s <seed></code>	伪随机数生成器的种子值。如果用相同的种子值再次运行 Monkey，它将生成相同的事件序列 举例 <code>adb shell monkey -s1111-v10</code>
<code>--throttle < 毫秒数 ></code>	在事件之间插入固定延迟。通过这个选项可以减缓 Monkey 的执行速度。如果不指定该选项，Monkey 将不会被延迟，事件将尽可能地地被生成
<code>--pct-touch < 百分比 ></code>	调整触摸事件的百分比（触摸事件是一个 <code>down-up</code> 事件，它发生在屏幕上的某单一位置）
<code>--pct-motion < 百分比 ></code>	调整动作事件的百分比（动作事件由屏幕上某处的一个 <code>down</code> 事件、一系列的伪随机事件和一个 <code>up</code> 事件组成）
<code>--pct-pinchzoom < 百分比 ></code>	调整二指缩放事件的百分比（二指缩放事件即智能机上的放大缩小手势操作）
<code>--pct-trackball < 百分比 ></code>	调整轨迹事件的百分比（轨迹事件由一个或几个随机的移动组成，有时还伴随点击）
<code>--pct-rotation < 百分比 ></code>	调整屏幕旋转事件的百分比（横屏和竖屏）
<code>--pct-nav < 百分比 ></code>	调整“基本”导航事件的百分比（导航事件由来自方向输入设备的 <code>up</code> 、 <code>down</code> 、 <code>left</code> 、 <code>right</code> 组成）
<code>--pct-majornav < 百分比 ></code>	调整“主要”导航事件的百分比（这些导航事件通常引发图形界面中的动作，如 5-way 键盘的中间按键、回退按键、菜单按键）
<code>--pct-syskeys < 百分比 ></code>	调整“系统”按键事件的百分比（这些按键通常被保留，由系统使用，如 Home、Back、Start Call、End Call 及音量控制键）
<code>--pct-appswitch < 百分比 ></code>	调整启动 Activity 的百分比（在随机间隔里，Monkey 通过调用 <code>startActivity</code> 方法最大限度地开启该 <code>package</code> 下的全部 Activity 的一种方法）
<code>--pct-flip < 百分比 ></code>	调整键盘事件的百分比（键盘事件如点击输入框、键盘弹起、点击输入框以外区域、键盘收回等）
<code>--pct-anyevent < 百分比 ></code>	调整其他类型事件的百分比（包罗了所有其他类型的事件，如按键、其他不常用的设备按钮等）

表4-3 Monkey约束限制类参数表

参数	说明
<code>-p < 包名 ></code>	如果用此参数指定了一个或几个包，Monkey 将只允许系统启动这些包里的 Activity。如果应用程序还需要访问其他包里的 Activity（如选择一个联系人），那些包也需要在此同时指定。如果不指定任何包，Monkey 将允许系统启动全部包里的 Activity。要指定多个包，需要使用多个 <code>-p</code> 选项，每个 <code>-p</code> 选项只能用于一个包
<code>-c < 类别名 ></code>	如果用此参数指定了一个或几个类别（Category），Monkey 将只允许系统启动被这些类别中的某个类别列出的 Activity。如果不指定任何类别，Monkey 将选择下列类别中列出的 Activity：Intent.CATEGORY_LAUNCHER 或 Intent.CATEGORY_MONKEY。要指定多个类别，需要使用多个 <code>-c</code> 选项，每个 <code>-c</code> 选项只能用于一个类别

4.调试类参数

通过调试类命令，可以对Monkey进行一些简单的调试，可以快速定位Monkey执行过程中的一些问题。如果用户想监控应用程序所调用的包之间的转换，则可以用--dbg-no-events参数；如果用户想监控内存泄漏，可以用--hprof参数。详细参数介绍见表4-4。

表4-4 Monkey调试类参数表

参数	说明
--dbg-no-events	设置此选项，Monkey 将执行初始启动，进入一个测试 Activity，不会再进一步生成事件。为了得到最佳结果，把它与 -v、一个或几个包约束，以及一个保持 Monkey 运行 30 秒或更长时间的非零值联合起来，从而提供一个可以监视应用程序所调用的包之间的转换的环境
--hprof	设置此选项，将在 Monkey 事件执行之前和执行之后生成内存快照文件存放于手机的 data/misc 目录。通过对比执行前后的内存快照文件，可以协助定位内存泄漏问题。由于内存快照文件比较大，所以要小心使用
--ignore-crashes	通常，当应用程序崩溃或发生任何失控异常时，Monkey 将停止运行。如果设置此选项，Monkey 将继续向系统发送事件，直到计数完成
--ignore-timeouts	通常，当应用程序发生任何超时错误（如出现“Application Not Responding”对话框）时，Monkey 将停止运行。如果设置了此选项，Monkey 将继续向系统发送事件，直到计数完成
--ignore-security-exceptions	通常，当应用程序发生许可错误（如启动一个需要某些许可的 Activity），Monkey 将停止运行。如果设置了此选项，Monkey 将继续向系统发送事件，直到计数完成
--kill-process-after-error	通常，当 Monkey 由于一个错误而停止时，出错的应用程序将继续处于运行状态。当设置了此选项时，将会通知系统停止发生错误的进程
--monitor-native-crashes	监视并报告 Android 系统中本地代码的崩溃事件
--wait-dbg	停止执行中的 Monkey，直到有调试器和它相连接

5.官方隐藏类参数

在Android官网上还有三个参数是看不到说明的，即为隐藏参数，这三个参数的详细介绍见表4-5。

表4-5 Monkey官方隐藏类参数表

参数	说明
<code>--pkg-blacklist-file < 黑名单文件 ></code>	限制 Monkey 不测试于指定黑名单文档中记录的包（ Package ）。若没有使用这个参数，Monkey 会测试系统内所有的包。若使用了该参数，可通过在黑名单文档内记录所有不要测试的包名称，来限制 Monkey 的执行范围。黑名单文档中每一行只能放一个包名
<code>--pkg-whitelist-file < 白名单文件 ></code>	限制 Monkey 只测试于指定的白名单文档中记录的包（ Package ）。若没有使用这个参数，Monkey 会测试系统内所有的包。若使用了该参数，可通过在白名单文档内记录所有要测试的包名，来限制 Monkey 的执行方位。白名单文档中每一行只能放一个包名 注意：若要测试的包与其他的包有关联，则必须一起指定这些包来执行这项参数
<code>-f < 脚本文件 ></code>	指定 Monkey 执行用户自定义的脚本文件 (在 4.2.1 节中会介绍改参数的详细使用方法)

4.1.3 Monkey事件

Monkey所执行的随机事件流中包含11大事件，分别是触摸事件、手势事件、二指缩放事件、轨迹事件、屏幕旋转事件、基本导航事件、主要导航事件、系统按键事件、启动Activity事件、键盘事件、其他类型事件。Monkey通过这11大事件来模拟用户的常规操作，对手机App进行稳定性测试。下面让我们来详细了解这11大事件。

1.触摸事件

触摸事件是指在屏幕某处按下并抬起的操作，可通过--pct-touch参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Sending Touch (ACTION_DOWN): 0,(444.0,1716.0)
:Sending Touch (ACTION_UP): 0,(447.18365,1728,0087)
```

该事件由一组Touch（ACTION_DOWN）和Touch（ACTION_UP）事件组成，在手机上看到实际操作类似于点击。

2.手势事件

手势事件是指在屏幕某处的按下、随机移动、抬起的操作，即直线滑动操作。可通过`--pct-motion`参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Sending Touch (ACTION_DOWN): 0:(282.0,750.0)
:Sending Touch (ACTION_MOVE): 0:(281.0507,745.5253)
:Sending Touch (ACTION_MOVE): 0:(274.9443,743.3276)
:Sending Touch (ACTION_MOVE): 0:(269.18774,738.50525)
:Sending Touch (ACTION_MOVE): 0:(260.14917,733.6212)
:Sending Touch (ACTION_UP): 0:(254.1414,730.6132)
```

该事件是由一个ACTION_DOWN事件、一系列ACTION_MOVE事件和一个ACTION_UP事件组成的，在手机上看到的实际操作是一个没有拐弯的直线滑动操作。

3.二指缩放事件

二指缩放事件是指在屏幕上的两处同时按下，并同时移动，最后同时抬起的操作，即智能机上的放大缩小手势操作。可通过`--pct-pinchzoom`参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Sending Touch (ACTION_DOWN): 0:(274.0,193.0)
:Sending Touch (ACTION_POINTER_DOWN 1): 0:(272.80875,198.17978) 1:(26.0,312.0)
:Sending Touch (ACTION_MOVE): 0:(251.31396,198.5104) 1:(24.973522,308.64676)
:Sending Touch (ACTION_MOVE): 0:(240.28494,202.44012) 1:(23.442032,307.8576)
:Sending Touch (ACTION_MOVE): 0:(221.90855,206.75597) 1:(22.903313,306.47507)
:Sending Touch (ACTION_MOVE): 0:(210.28592,212.24286) 1:(17.78174,303.11304)
:Sending Touch (ACTION_POINTER_UP 1): 0:(171.06334,236.1724) 1:
(10.3147135,293.79877)
:Sending Touch (ACTION_UP): 0:(161.06638,240.22447)
```

该事件起始是一个ACTION_DOWN事件和一个ACTION_POINTER_DOWN事件，即模拟两个手指同时点下；中间是一系列的ACTION_MOVE事件，即两个手指同时在屏幕上直线滑动；结束是由一个ACTION_POINTER_UP事件和一个ACTION_UP事件组成的，即两个手指同时放开。

4. 轨迹事件

轨迹事件是由一个或多个随机的移动组成的，有时会伴随着点击。很早之前的Android手机带有轨迹球，这个事件就是模拟的轨迹球的操作。现在的手机几乎都没有轨迹球，但轨迹球事件中包含曲线滑动操作，如果被测程序需要曲线滑动时可以选用此参数。可通过--pct-trackball参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Sending Trackball (ACTION_MOVE): 0:(2.0,3.0)
:Sending Trackball (ACTION_MOVE): 0:(-1.0,4.0)
:Sending Trackball (ACTION_MOVE): 0:(2.0,-3.0)
```

该事件是由一系列的Trackball（ACTION_MOVE）事件组成的，观察手机上的操作，即为一系列的曲线滑动操作。

5. 屏幕旋转事件

屏幕旋转事件是一个隐藏事件，在Android官方文档中并没有记录这个事件。它其实是模拟的Android手机的横屏和竖屏切换。可通过--pct-rotation参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Sending rotation degree=1, persist=false
:Sending rotation degree=3, persist=true
:Sending rotation degree=2, persist=true
:Sending rotation degree=0, persist=true
```

该事件由一个rotation事件组成，其中degree表示的是旋转方向，顺时针旋转，0表示旋转90度的方向，1表示旋转180度的方向，2表示旋转270度的方向，3表示旋转360度的方向。在执行过程中，可以看到手机屏幕在横竖屏之间不断地切换。

6.基本导航事件

基本导航事件是指点击方向输入设备的上、下、左、右按键的操作，现在手机上很少有上、下、左、右按键，这种事件一般用得比较少。可通过--pct-nav参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Sending Key (ACTION_DOWN): 19 // KEYCODE_DPAD_UP
:Sending Key (ACTION_UP): 19 // KEYCODE_DPAD_UP
:Sending Key (ACTION_DOWN): 20 // KEYCODE_DPAD_DOWN
:Sending Key (ACTION_UP): 20 // KEYCODE_DPAD_DOWN
:Sending Key (ACTION_DOWN): 21 // KEYCODE_DPAD_LEFT
:Sending Key (ACTION_UP): 21 // KEYCODE_DPAD_LEFT
:Sending Key (ACTION_DOWN): 22 // KEYCODE_DPAD_RIGHT
:Sending Key (ACTION_UP): 22 // KEYCODE_DPAD_RIGHT
```

该事件是由一个Key（ACTION_DOWN）和一个Key（ACTION_UP）组成的，点击的就是上、下、左、右四个方向按键。

7.主要导航事件

主要导航事件是指点击“主要导航”按键的操作，这些按键通常会导致UI界面中的动作，如5-way键盘的中间键、回退按键、菜单按键。可通过--pct-majornav参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Sending Key (ACTION_DOWN): 23    // KEYCODE_DPAD_CENTER
:Sending Key (ACTION_UP): 23      // KEYCODE_DPAD_CENTER
:Sending Key (ACTION_DOWN): 82    // KEYCODE_MENU
:Sending Key (ACTION_UP): 82      // KEYCODE_MENU
```

该事件是由一个Key（ACTION_DOWN）和一个Key（ACTION_UP）组成的，点击的按键就是中间键和菜单键。

8.系统按键事件

系统按键事件是指点击系统保留使用的按键的操作，如点击Home键、返回键、音量调节键等。可通过--pct-syskeys参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Sending Key (ACTION_DOWN): 5    // KEYCODE_CALL
:Sending Key (ACTION_UP): 5      // KEYCODE_CALL
```

```
:Sending Key (ACTION_DOWN): 4 // KEYCODE_BACK
:Sending Key (ACTION_UP): 4 // KEYCODE_BACK
:Sending Key (ACTION_DOWN): 3 // KEYCODE_HOME
:Sending Key (ACTION_UP): 3 // KEYCODE_HOME
:Sending Key (ACTION_DOWN): 24 // KEYCODE_VOLUME_UP
:Sending Key (ACTION_UP): 24 // KEYCODE_VOLUME_UP
:Sending Key (ACTION_DOWN): 25 // KEYCODE_VOLUME_DOWN
:Sending Key (ACTION_UP): 25 // KEYCODE_VOLUME_DOWN
```

该事件是由一个Key（ACTION_DOWN）和一个Key（ACTION_UP）组成的，点击的就是上面说到的几个系统按键。

9.启动Activity事件

启动Activity事件是指在手机上启动一个Activity的操作。在随机的时间间隔中，Monkey将执行一个startActivity（）方法，作为最大限度上覆盖被测包中全部Activity的一种方法。可通过--pct-appswitch参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x10200000;component=com.android.settings/.Settings;end
// Allowing start of Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.android.settings/.Settings } in package com.android.settings
```

该事件是由一个Switch操作组成的，从手机上看，上面的操作实际是打开了com.android.settings这个应用的一个com.android.settings.Settings的Activity界面。

10.键盘事件

键盘事件主要是一些与键盘相关的操作。比如点击输入框、键盘弹起、点击输入框以外区域、键盘收回等。可通过--pct-flip参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Sending Flip keyboardOpen=false
:Sending Flip keyboardOpen=true
```

如日志所示，这里主要是键盘的打开和关闭操作。

11.其他类型事件

其他类型事件包括了除前面提到的10种事件外其他所有的事件，如按键、其他不常用的设备上的按钮等。可通过--pct-anyevent参数来配置其事件百分比。从Monkey执行该事件对外输出的日志可以看到：

```
:Sending Key (ACTION_DOWN): 59      // KEYCODE_SHIFT_LEFT
:Sending Key (ACTION_UP): 59        // KEYCODE_SHIFT_LEFT
:Sending Key (ACTION_DOWN): 138     // KEYCODE_F8
:Sending Key (ACTION_UP): 138      // KEYCODE_F8
:Sending Key (ACTION_DOWN): 45      // KEYCODE_Q
:Sending Key (ACTION_UP): 45        // KEYCODE_Q
:Sending Key (ACTION_DOWN): 192     // KEYCODE_BUTTON_5
:Sending Key (ACTION_UP): 192      // KEYCODE_BUTTON_5...
```

该事件是由一个Key（ACTION_DOWN）和一个Key（ACTION_UP）组成的，点击的按键就是其他的一些系统按键，如字母按键、数字按键等。因为现在手机很少带字母按键或数字按键，所以这个事件一般使用得比较少。

4.1.4 Monkey环境搭建

了解了Monkey的参数及11大事件后，接下来让我们了解一下Monkey的运行环境搭建方法。

Monkey是由adb命令来启动的，故只要配置好adb环境即可。

(1) 下载并安装Android SDK和JDK。

(2) 将Android SDK目录下的platform-tools和tools目录配置到系统环境变量Path中。

(3) 打开cmd命令行窗口，输入“adb”，能显示adb帮助信息，如图4-2所示，则Monkey环境配置成功。

```
管理員: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7600]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\sharonzheng>adb
Android Debug Bridge version 1.0.31

-a                - directs adb to listen on all interfaces for a connection
-d                - directs command to the only connected USB device
                  returns an error if more than one USB device is present.
-e                - directs command to the only running emulator.
                  returns an error if more than one emulator is running.
-s <specific device> - directs command to the device or emulator with the given
                  serial number or qualifier. Overrides ANDROID_SERIAL
                  environment variable.
-p <product name or path> - simple product name like 'sooner', or
                  a relative/absolute path to a product
                  out directory like 'out/target/product/sooner'.
                  If -p is not specified, the ANDROID_PRODUCT_OUT
                  environment variable is used, which must
                  be an absolute path.
-H                - Name of adb server host (default: localhost)
-P                - Port of adb server (default: 5037)
devices [-l]      - list all connected devices
                  ('-l' will also list device qualifiers)
connect <host>[:<port>] - connect to a device via TCP/IP
                  Port 5555 is used by default if no port number is specified
disconnect [<host>[:<port>]] - disconnect from a TCP/IP device.
                  Port 5555 is used by default if no port number is specified
                  Using this command with no additional arguments
```

图4-2 Monkey环境配置成功

4.1.5 Monkey启动

Monkey启动方式很简单：先连接被测手机到PC上，然后打开CMD命令行窗口输入对应的adb命令行即可。通过命令行启动Monkey有两种方式：

·直接PC启动

```
> adb shell monkey [options] <count>
```

·Shell端启动

```
>adb shell  
>monkey [options] <count>
```

这两者的区别是，通过PC端启动，Monkey运行日志可以保存在PC上；通过Shell端启动，Monkey运行日志可以保存在手机里（保存结果的命令在4.2.2节会详细介绍）。



注意 Monkey启动后会不断地向被测对象发送随机事件流，直到事件执行完毕或者发生异常时才停止。在Monkey运行过程中，即便断开与PC的连接，Monkey依然可以在手机上继续运行。

停止Monkey的方法是：直接杀掉手机上的Monkey进程。具体方法如下：

```
>adb shell ps |grep monkey
```

获取到com.android.commands.monkey的进程ID

```
>adb shell kill pid
```

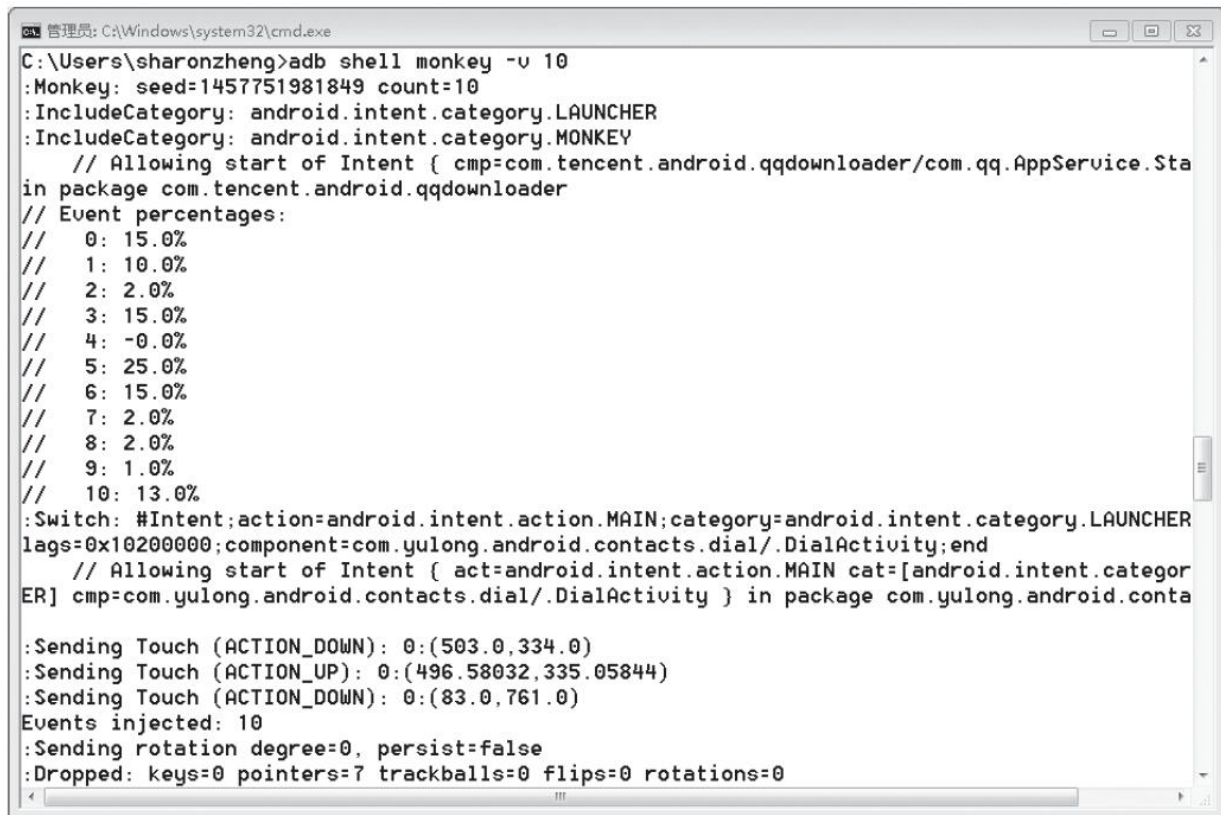
举例：adb shell kill 30898

通过kill命令杀死对应的Monkey进程。

下面来看一个最简单的Monkey命令行示例：

```
> adb shell monkey -v 10
```

通过该命令启动Monkey后，Monkey向被测手机的Android系统发送10条随机事件流。当启动运行Monkey测试后，手机上会开始执行Monkey测试，同时在命令行窗口输出日志，执行完成后，可以看到如图4-3所示的日志信息。



```
C:\Users\sharonzheng>adb shell monkey -v 10
:Monkey: seed=1457751981849 count=10
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
// Allowing start of Intent { cmp=com.tencent.android.qqdownloader/com.qq.AppService.Sta
in package com.tencent.android.qqdownloader
// Event percentages:
// 0: 15.0%
// 1: 10.0%
// 2: 2.0%
// 3: 15.0%
// 4: -0.0%
// 5: 25.0%
// 6: 15.0%
// 7: 2.0%
// 8: 2.0%
// 9: 1.0%
// 10: 13.0%
:Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER
lags=0x10200000;component=com.yulong.android.contacts.dial/.DialActivity;end
// Allowing start of Intent { act=android.intent.action.MAIN cat=[android.intent.categor
ER] cmp=com.yulong.android.contacts.dial/.DialActivity } in package com.yulong.android.conta
:Sending Touch (ACTION_DOWN): 0:(503.0,334.0)
:Sending Touch (ACTION_UP): 0:(496.58032,335.05844)
:Sending Touch (ACTION_DOWN): 0:(83.0,761.0)
Events injected: 10
:Sending rotation degree=0, persist=false
:Dropped: keys=0 pointers=7 trackballs=0 flips=0 rotations=0
```

图4-3 Monkey执行的日志输出

4.2 Monkey测试方法

了解了**Monkey**的命令行参数、11大用户事件、运行环境和命令行使用方法后，这一节将重点介绍**Monkey**测试在实际业务中的运用方法，这里包括常见的几种测试方法以及**Monkey**日志分析方法。

4.2.1 Monkey测试实例

前面介绍了Monkey的一些基础知识，这一节主要介绍Monkey测试在实际项目中如何运用。

1. 常规的稳定性测试

测试背景：

这是一个海外的合作项目，被测程序是Android应用（App）。测试希望通过Monkey来模拟用户长时间的随机操作，检查被测应用是否会出现异常（应用崩溃或者无响应）。

测试脚本：

```
adb shell monkey -p com.xxx.xxx --pct-touch 40 --pct-motion 25 --pct-appswitch 10 --pct-rotation 5 -s 12358 --throttle 400 --ignore-crashes --ignore-timeouts -v 500000
```

显而易见，这个Monkey测试的命令相比上一节提到的要复杂得多，主要是对一些操作事件做了限制，从而减少了Monkey伪随机化的无效操作。这体现在以下几个方面。

- 1) 使用-p参数来制定测试应用的包名（Package）

因为被测程序是一个特定的Android应用程序，需要指定被测程序的包名。指定包名后，Monkey会根据包名找到对应的应用，并启动其main activity，然后执行Monkey测试。



技巧 查找应用包名的方法有很多，这里简单列举几个常用方法：

(1) 通过pm命令查看。

在命令行窗口输入：

```
>adb shell
>pm list package
```

此时将列出手机上所有的应用包名，在列表中找到要测试的应用包名即可。

(2) 通过查看APK源码下的AndroidManifest.xml文件。

(3) 通过aapt命令查看。

(4) 通过adb logcat抓取当前Android机运行的App的包名。

2) 使用--pct-xxx参数限制Monkey执行的事件类型和占比

前面已经说了，这个测试的目的是希望模拟用户操作，因此需要让Monkey执行的事件尽可能地接近用户的常规操作，这样才可以最大

限度地发现用户使用过程中可能出现的问题。因此需要对Monkey执行的事件百分比做一些调整。

触摸事件和手势事件是用户最常见的操作，所以通过--pct-touch和--pct-motion将这两个事件的占比调整到40%与25%；目标应用包含了多个Activity，为了能覆盖大部分的Activity，所以通过--pct-appswitch将Activity切换的事件占比调整到10%；被测应用之前在测试中出现过不少横竖屏之间切换的问题，这个场景也必须关注，因此通过--pct-rotation把横竖屏切换事件调整到10%。

3) 使用-s参数来指定命令执行的seed值

Monkey会根据seed值来生成对应事件流，同一个seed生成的事件流是完全相同的。这里指定了seed值，是为了测试发现问题时，便于进行问题复现。

4) 使用--throttle参数来控制Monkey每个操作之间的时间间隔

指定操作之间的时间间隔，一方面是希望能更接近用户的操作场景，正常用户操作都会有一定的时间间隔；另一方面也是不希望因为过于频繁的操作而导致系统崩溃，尤其是在比较低端的手机上执行测试时。因此通过--throttle设置Monkey每个操作固定延迟0.4秒。

5) 使用--ignore-crash和--ignore-timeouts参数使Monkey遇到意外时能继续执行

在执行Monkey测试时，会因为应用的崩溃或没有响应而意外终止，所以需要在命令中增加限制参数--ignore-crash和--ignore-timeouts，让Monkey在遇到崩溃或没有响应的时候，能在日志中记录相关信息，并继续执行后续的测试。

6) 使用-v指定log的详细级别

Monkey的日志输出有3个级别：默认的是level 0，-v-v日志级别为level 1，-v-v-v日志级别为level 2。日志的级别越高，其详细程度也越高。为了方便问题的定位，将日志级别设置为level2。

在常规的稳定性测试中，虽然可以自定义各种事件的操作占比，但毕竟是随机事件流。在实际测试过程中，难免会遇到Monkey点了我们不希望它点击的地方，比如误点了工具栏导致网络断开的情况等。当测试过程中Wi-Fi断开时，是否有可能自动重连呢？在后面的章节中会介绍如何解决这个问题，大家可以带着这个问题继续往下看。

2.自定义脚本的稳定性测试

常规Monkey测试执行的是随机的事件流，但如果只是想让Monkey测试某个特定场景（执行固定的事件流）呢？这时候就需要用到自定

义脚本了，Monkey支持执行用户自定义脚本的测试，用户只需要按照Monkey脚本的规范编写好脚本，存放到手机上，启动Monkey通过-f scriptfile参数调用脚本即可。

Monkey自定义脚本的编写模板如代码清单4-1所示。

代码清单4-1 Monkey自定义脚本的编写模板

```
#头文件，控制

Monkey发送消息的参数，固定写即可

#脚本类型，一般不用更改

type=raw events
#脚本执行次数，但是由于

Monkey命令本身可以指定执行次数，所以这里的设置是不生效的

count=10
#命令执行速率，速率也可以通过

Monkey命令设置，这里的设置是不生效的

speed=1.0
#以下为

Monkey命令

start data>>
LaunchActivity (

pkg_name,cl_name)

DispatchPress(KEYCODE_HOME)...
```

Monkey脚本常见API见表4-6。

表4-6 Monkey脚本常见API

API	说明
LaunchActivity(Pkg_name,cl_name)	启动被测应用的某个 Activity Pkg_name: 包名 cl_name: Activity 名
Tap(x,y,tapDuration)	模拟一次手指单击事件 x: 点击的横坐标 y: 点击的纵坐标 tapDuration: 按下的时长, 单位是 ms
DispatchPress(keyName)	模拟按键点击 keyName: 按键的名称
RotateScreen(rotationDegree,peresist)	模拟旋转屏幕 rotationDegree: 用 0 ~ 3 分别表示顺时针旋转的四个方向 peresist: 是否存留
DispatchFlip(true/false)	打开或关闭软键盘
LongPress()	长按两秒
PressAndHold(x,y,pressDuration)	模拟长按事件: 即单指按下一段时间, 再抬起 x: 点击的横坐标 y: 点击的纵坐标 pressDuration: 点击时长, 单位是 ms
DispatchString(input)	输入字符串 input: 输入内容
Drag(xStart,yStart,xEnd,yEnd,stepCount)	模拟拖动操作: 即单指按下、拖动、放开 xStart、yStart: 起始的横坐标和纵坐标 xEnd、yEnd: 结束的横坐标和纵坐标 stepCount: 移动的事件数 (可理解为移动速度)

(续)

API	说明
PinchZoom(pt1xStart, pt1yStart, pt1xEnd, pt1yEnd, pt2xStart, pt2yStart, pt2xEnd, pt2yEnd, stepCount)	模拟缩放手势: 即两个手指同时按下并移动, 再同时放开 pt1xStart、pt1yStart: 手指 1 起始的横坐标和纵坐标 pt1xEnd、pt1yEnd: 手指 1 结束的横坐标和纵坐标 pt2xStart、pt2yStart: 手指 2 起始的横坐标和纵坐标 pt2xEnd、pt2yEnd: 手指 2 结束的横坐标和纵坐标 stepCount: 移动的事件数 (可理解为移动速度, step Count 越大移动速度越慢)
UserWait(sleepTime)	设置等待时间 sleepTime: 等待的时间, 单位是 ms
DeviceWakeUp()	唤醒屏幕



技巧

Monkey脚本只能通过坐标的方式来定位点击和移动事件的屏幕位置，这里就需要提前获取坐标信息。获取坐标信息的方法很多，最简单的方法就是打开手机中的开发人员选项，打开“显示指针位置”。随后，在屏幕上的每次操作，在导航栏上都会显示坐标信息。

下面来看一个简单的例子：

这里要测试的是应用宝App，测试的操作是打开应用宝，点击输入框，输入“yyb”，点击搜索。搜索完成后，点击返回键返回应用宝首页。

首先，将在本地编写需要的测试脚本命名为**monkey.script**（文件格式无要求），脚本内容如代码清单4-2所示。

代码清单4-2 Monkey自定义脚本实现进入应用宝进行搜索

```
#启动测试
```

```
type = user
count = 49
speed = 1.0
start data >>
#启动应用宝
```

```
LaunchActivity(com.tencent.android.qqdownloader,com.tencent.assistant.activity.SplashActivity)
UserWait(2000)
#点击搜索框
```

```
Tap(463,150,1000)
UserWait(2000)
#输入字母“
```

```
yyb”
```

```
DispatchString(yyb)
UserWait(2000)
#点击搜索
```

```
Tap(960,150,1000)
UserWait(2000)
#点击返回键返回首页
```

```
DispatchPress(KEYCODE_BACK)
```

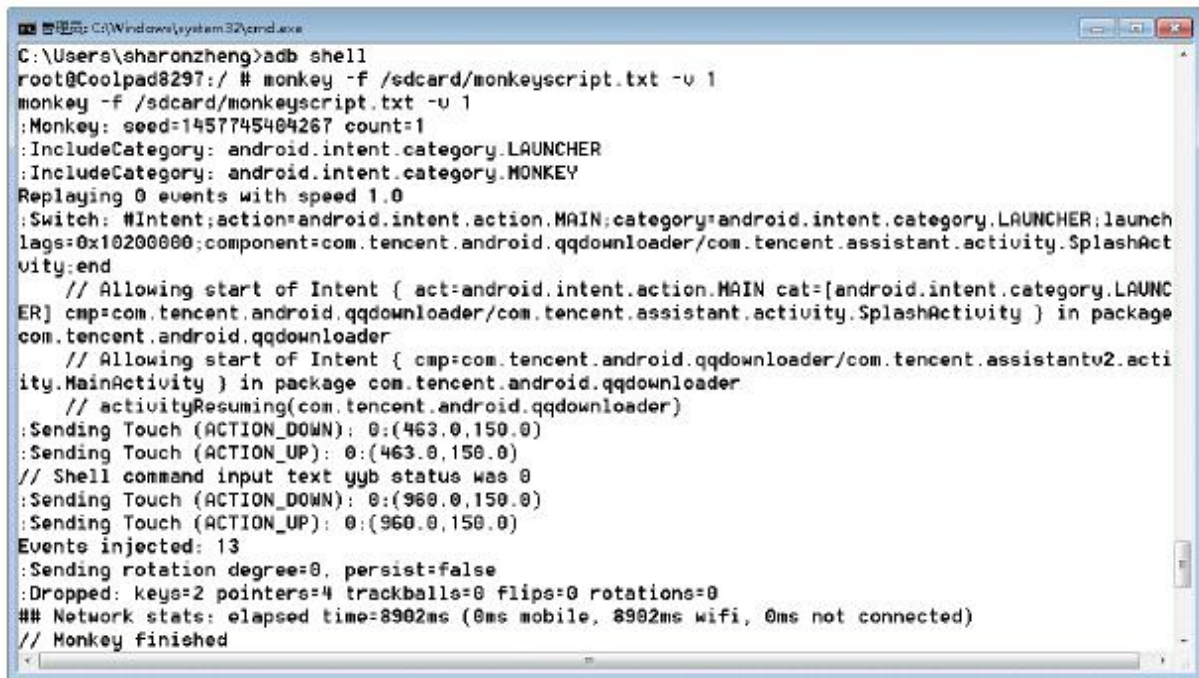
其次，将文件push到手机或模拟器的sdcard中：

```
>adb push monkey.script /sdcard/
```

最后，执行脚本：

```
>adb shell monkey -f /sdcard/monkey.script -
v 1
```

此时可以看到手机按照前面的脚本开始执行了，命令行窗口输出的结果如图4-4所示。



```
C:\Users\sharonzheng>adb shell
root@Coolpad8297:/ # monkey -f /sdcard/monkeysript.txt -v 1
monkey -f /sdcard/monkeysript.txt -v 1
:Monkey: seed=1457745404267 count=1
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
Replaying 0 events with speed 1.0
:Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launch
lags=0x10200000;component=com.tencent.android.qqdownloader/com.tencent.assistant.activity.SplashAct
vity:end
// Allowing start of Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNC
ER] cmp=com.tencent.android.qqdownloader/com.tencent.assistant.activity.SplashActivity } in package
com.tencent.android.qqdownloader
// Allowing start of Intent { cmp=com.tencent.android.qqdownloader/com.tencent.assistantv2.acti
vity.MainActivity } in package com.tencent.android.qqdownloader
// activityResuming(com.tencent.android.qqdownloader)
:Sending Touch (ACTION_DOWN): 0:(463.0,150.0)
:Sending Touch (ACTION_UP): 0:(463.0,150.0)
// Shell command input text yyb status was 0
:Sending Touch (ACTION_DOWN): 0:(960.0,150.0)
:Sending Touch (ACTION_UP): 0:(960.0,150.0)
Events injected: 13
:Sending rotation degree=0, persist=false
:Dropped: keys=2 pointers=4 trackballs=0 flips=0 rotations=0
## Network stats: elapsed time=8902ms (0ms mobile, 8902ms wifi, 0ms not connected)
// Monkey finished
```

图4-4 命令行窗口输出的结果


从手机上可以看到，Monkey按照脚本启动了应用宝，随后点击搜索框，输入“yyb”，点击搜索，如图4-5和图4-6所示。



图4-5 脚本启动应用宝



图4-6 点击搜索框，搜索“yyb”

 **窍门** 如果需要重复执行某个脚本，只要在Monkey启动命令中修改执行次数即可。例如：

```
adb shell monkey -f /sdcard/monkey.script -v 10
```

此时Monkey会重复执行monkey.script脚本10次。

3.结合辅助命令，获取更多信息

常规测试只要记录下Monkey日志，再分析Monkey日志检查是否有异常即可。但是，很多时候，测试除了想知道执行过程是否有异常，还需要能获取执行过程中的一些详细信息或性能数据，比如想知道Monkey执行过程中是否存在内存泄漏，需要获取内存信息。这时候就需要借助一些辅助的命令来获取更多信息了。下面列举了几种Monkey测试中常用的辅助命令，使用方法也非常简单，只要在执行Monkey的同时，另起一个CMD命令行窗口输入对应命令执行即可。

·获取logcat日志信息:

```
adb shell logcat -v time>log.txt
```

·获取内存信息:

```
adb shell dumpsys meminfo <进程名>
```

·获取CPU消耗信息:

```
adb shell top -n 1 |find "进程名"
```

·获取电量信息:

```
adb shell dumpsys battery
```

·获取GPU信息:

GPU信息命令:

```
adb shell dumpsys gfxinfo <进程名>  
>
```

·获取流量信息:

```
adb shell cat /proc/uid_stat/<被测应用的  
uid>/tcp_rcv
```



技巧 如何获取被测应用的UID

步骤1: 查看被测应用的进程ID (PID)

```
adb shell ps | grep <被测应用包名>  
>
```

步骤2: 查看被测应用的用户ID (UID)

```
adb shell cat /proc/$pid/status
```

4.Monkey测试策略制定思路

前面介绍了几种常见的Monkey测试方法，但在实际项目中，选择哪种Monkey测试策略，则需要根据实际项目的情况来做判断。主要是看测试目的及被测应用自身的特点。假如我们想测试浏览器的双指缩

放功能是否有异常，那就需要选择--pct-pinchzoom参数，调大双指缩放事件的占比进行Monkey测试；假如我们想验证ROM的横竖屏切换功能是否正常，那就需要选择--pct-rotation参数，调大横竖屏切换事件的占比进行Monkey测试；假如我们想验证重复某种特定操作时，应用是否存在异常，那可以选择-f参数，自定义Monkey脚本进行验证；假如我们想验证长时间操作时应用是否存在内存泄漏，那就需要结合-hprof参数和dumpsys meminfo<进程名>进行Monkey测试。

总之，Monkey测试策略是需要依据测试目的和被测程序的特点来制定的。

4.2.2 Monkey日志分析

Monkey日志分析是Monkey测试中非常重要的一个环节，通过日志分析，可以获取当前测试对象在测试过程中是否会发生异常，以及发生的概率，同时还可以获取对应的错误信息，帮助开发定位和解决问题。介绍日志分析方法之前，先来看一下日志的保存方法。

1.Monkey日志的保存方法

Monkey运行日志常见的保存方法有三种：

·保存在PC中，代码如下：

```
>adb shell monkey [option] <count> >d:\monkey.txt
```

执行以上命令，Monkey的运行日志将被保存在PC上的D盘下的一个monkey.txt文件中。

·保存在手机中，代码如下：

```
>adb shell  
>monkey [option] <count> > /mnt/sdcard/monkey.txt
```

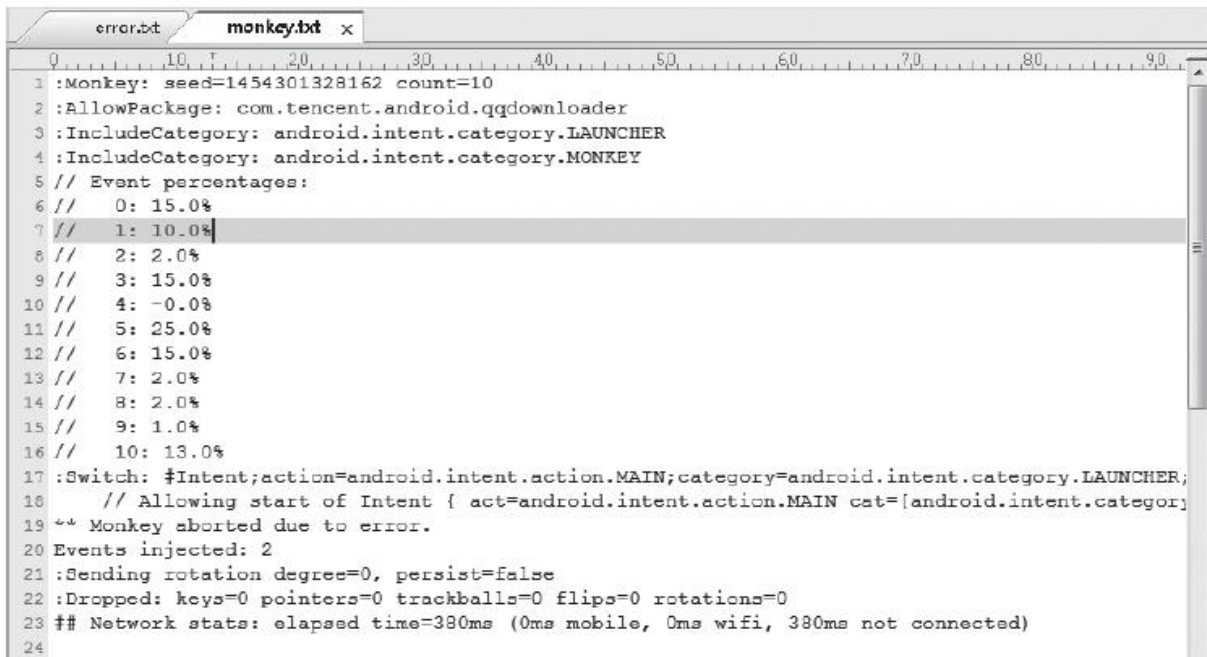
执行以上命令，Monkey的运行日志将被保存在手机中的SD卡上的一个monkey.txt文件中。

·标注流与错误流分开保存，代码如下：

```
Monkey [option] <count> 1>/sdcard/monkey.txt 2>/sdcard/error.txt
```

执行以上命令，Monkey的运行日志和异常日志将被分开保存。此时Monkey的运行日志将被保存在monkey.txt文件中，而异常日志将被保存在D盘下的error.txt中。

执行结束后，可以看到SD卡上新增加了monkey.txt和error.txt。monkey.txt显示运行日志，如图4-7所示。



```
1 :Monkey: seed=1454301328162 count=10
2 :AllowPackage: com.tencent.android.qqdownloader
3 :IncludeCategory: android.intent.category.LAUNCHER
4 :IncludeCategory: android.intent.category.MONKEY
5 // Event percentages:
6 // 0: 15.0%
7 // 1: 10.0%
8 // 2: 2.0%
9 // 3: 15.0%
10 // 4: -0.0%
11 // 5: 25.0%
12 // 6: 15.0%
13 // 7: 2.0%
14 // 8: 2.0%
15 // 9: 1.0%
16 // 10: 13.0%
17 :Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;
18 // Allowing start of Intent { act=android.intent.action.MAIN cat=[android.intent.category
19 +- Monkey aborted due to error.
20 Events injected: 2
21 :Sending rotation degree=0, persist=false
22 :Dropped: keys=0 pointers=0 trackballs=0 flips=0 rotations=0
23 ## Network stats: elapsed time=380ms (0ms mobile, 0ms wifi, 380ms not connected)
24
```

图4-7 运行日志输出

如果Monkey执行期间存在Crash（崩溃）或ANR（Application Not Responding，应用程序无响应），error.txt中会显示错误日志，如图4-8

所示。

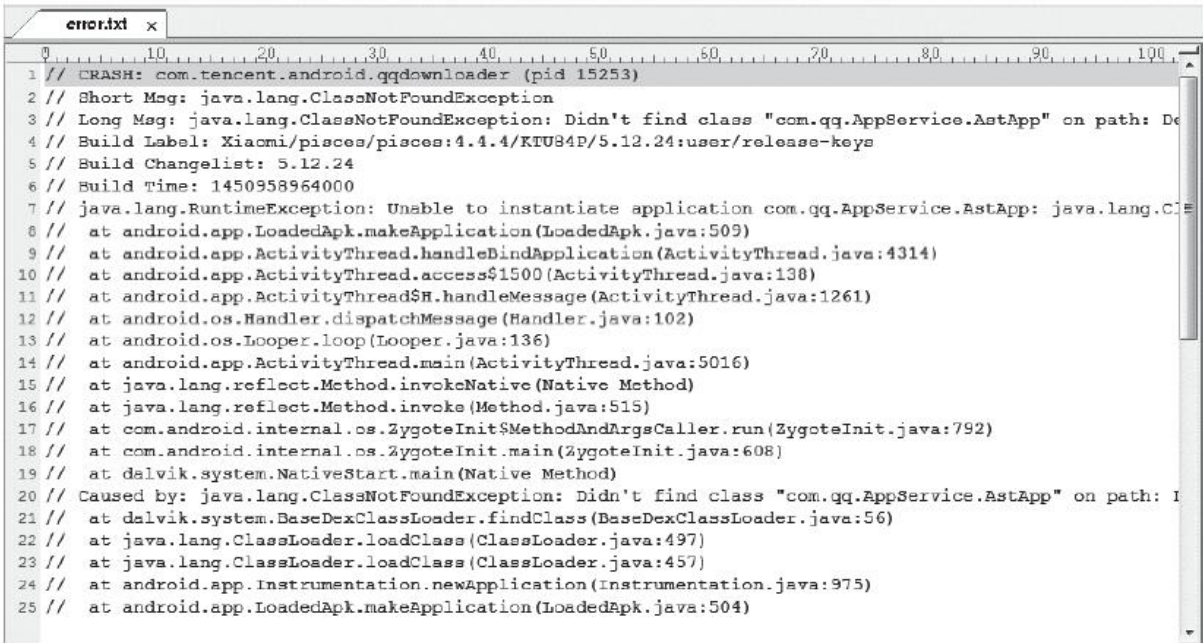


图4-8 异常日志输出

2.Monkey日志内容解析

Monkey运行时输出的日志一般包含四类信息，分别是测试命令信息、伪随机事件流信息、异常信息、Monkey执行结果信息。

1) 测试命令信息

Monkey启动后会输出当前所执行命令的各种参数信息，其中包括种子（Seed）信息、事件数量、可运行的应用列表以及各事件百分比等。这些信息都是通过Monkey命令参数所指定的，这部分日志信息的解析，如代码清单4-3所示。

代码清单4-3 Monkey日志-测试命令信息

```
// 测试命令信息

// 随机种子值，执行事件数量

:Monkey: seed=1454215444564 count=10
// 可运行的应用列表

:AllowPackage: com.tencent.android.qqdownloader
// Category包含

LAUNCHER和

MONKEY
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
// 各事件的百分比

// Event percentages:
//   0: 15.0%   事件

0:

--pct-touch
//   1: 10.0%   事件

1:

--pct-motion
//   2: 2.0%   事件

2:

--pct-pinchzoom
//   3: 15.0%   事件

3:

--pct-trackball
//   4: -0.0%   事件

4:

--pct-rotation
//   5: 25.0%   事件

5:

--pct-nav
//   6: 15.0%   事件

6:
```

```
--pct-majornav
// 7: 2.0% 事件

7:

--pct-syskeys
// 8: 2.0% 事件

8:

--pct-appswitch
// 9: 1.0% 事件

9:

--pct-flip
// 10: 13.0% 事件

10:

--pct-anyevent
```

2) 伪随机事件流信息

当Monkey开始执行测试后，会顺序输出执行的事件流信息，主要是前面提到的11大事件。这部分日志信息的解析，如代码清单4-4所示。

代码清单4-4 Monkey日志—伪随机事件流信息

```
//执行的事件流信息

//启动

App事件

:Switch:
#Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x10200000;component=com.tencent.android.qqdownloader/com.tencent.assistant.activity.SplashActivity;end
// Allowing start of Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
cmp=com.tencent.android.qqdownloader/com.tencent.assistant.activity.SplashActivity
} in packagecom.tencent.android.qqdownloader
//轨迹球事件
```

```
:Sending Trackball (ACTION_MOVE): 0:(4.0,2.0)
//点击事件

:Sending Touch (ACTION_DOWN): 0:(387.0,1858.0)
:Sending Touch (ACTION_UP): 0:(385.8215,1861.3011)
//延时

Sleeping for 0 milliseconds...
```

3) 异常信息

当Monkey执行过程中遇到错误时，会输出对应异常信息，如代码清单4-5所示。

代码清单4-5 Monkey日志-异常信息

```
//发送

Crash的应用包名和

pid
// CRASH: com.tencent.android.qqdownloader (pid 912)
//Crash的简要信息

// Short Msg: java.lang.ClassNotFoundException
//Crash的详细信息

// Long Msg: java.lang.ClassNotFoundException: Didn't find class "com.
qq.AppService.AstApp" on path DexPathList[[zip file "/data/app/com.tencent.
android.qqdownloader-2.apk"],nativeLibraryDirectories[/data/app-lib/com.
tencent.android.qqdownloader-2, /vendor/lib, /system/lib]]
//机型和系统信息

// Build Label: Xiaomi/pisces/pisces:4.4.4/KTU84P/5.12.24:user/release-keys
// Build Changelist: 5.12.24
// Build Time: 1450958964000
//Crash的详细日志

// java.lang.RuntimeException: Unable to instantiate application com.
qq.AppService.AstApp: java.lan.ClassNotFoundException: Didn't find class "com.
qq.AppService.AstApp" on path: DexPathList[[zip fil "/data/app/com.tencent.
android.qqdownloader-2.apk"],nativeLibraryDirectories=[/data/app-lib/com.
```



```
tecent.android.qqdownloader-2, /vendor/lib, /system/lib]]
//      at android.app.LoadedApk.makeApplication(LoadedApk.java:509)
//      at android.app.ActivityThread.access$1500(ActivityThread.java:138)
//      at dalvik.system.NativeStart.main(Native Method)
//      ... 11 more
//
```

4) Monkey执行结果信息

当Monkey执行完所有事件后，会输出执行结果信息，其中包括执行的事件数量、旋转的角度、丢失的事件数量、网络状态以及Monkey最终的执行结果，如代码清单4-6所示。

代码清单4-6 Monkey日志—执行成功结果信息

```
//执行的事件数量

Events injected: 10
//旋转的角度为

0
:Sending rotation degree=0, persist=false
//丢失的事件数量

:Dropped: keys=0 pointers=0 trackballs=0 flips=0 rotations=0
//网络状态，移动网络联网

0ms,

Wi-Fi联网

0ms, 没联网

144ms
## Network stats: elapsed time=144ms (0ms mobile, 0ms wifi, 144ms not connected)
// Monkey finished
```

如果Monkey执行过程中出现了异常导致执行失败，会输出对应的执行失败的原因，第几个事件执行失败以及所使用的随机种子数，如

代码清单4-7所示。

代码清单4-7 Monkey日志-执行失败结果信息

```
//显示
Monkey执行失败

** Monkey aborted due to error.
//执行的事件数量

Events injected: 8
//旋转的角度为

0
:Sending rotation degree=0, persist=false
//丢失的事件数量

:Dropped: keys=0 pointers=0 trackballs=0 flips=0 rotations=0
//网络状态

## Network stats: elapsed time=405ms (0ms mobile, 0ms wifi, 405ms not connected)
//提示在执行到第

8个事件时出现

Crash, 以及所使用的随机种子的值

** System appears to have crashed at event 8 of 100 using seed 1454216848235
```

3.Monkey日志异常信息查找

Monkey执行过程中常见的错误类型主要有两类：应用程序无响应（ANR）和崩溃（Crash）。

ANR是指当Android系统监测到应用程序在5秒内没有响应输入的事件或广播在10秒内没有执行完毕时抛出无响应提示。当出现ANR时

弹出的错误提示框如图4-9所示。

Crash是指当应用程序出现错误时导致程序异常停止或退出的情况，当出现Crash时通常会弹出对应的错误提示框如图4-10所示。

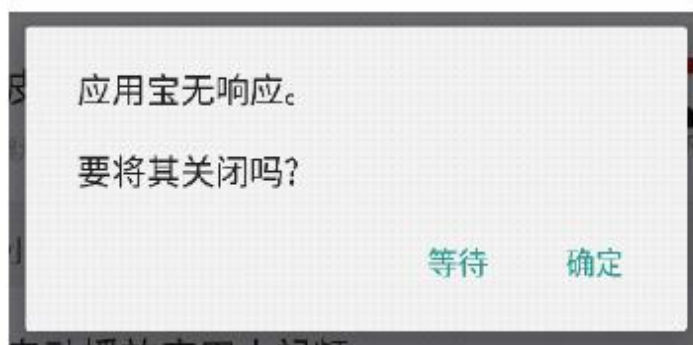


图4-9 ANR弹窗

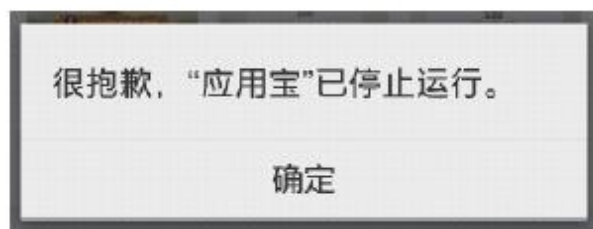


图4-10 Crash弹窗

要统计Monkey日志中错误出现的次数也非常简单，只要搜索关键字“ANR”和“CRASH”出现的次数即可。由于通常Monkey测试的日志会比较大，日志内容也非常多，为了简化统计操作，可以使用bat脚本进行统计，具体如代码清单4-8所示。

代码清单4-8 Monkey日志分析bat脚本

```
@echo off&setlocal enabledelayedexpansion
#设置所有
```

Monkey日志存放的目录

```
set ff=log\*.txt
#设置查询关键字
```

```
set str=CRASH crash ANR died
#设置查询结果存放的目录
```

```
set fileName=Result.txt
#开始查询
```

echo 正在统计

```
&echo;
echo %date% %time% >%fileName%
echo.>>%fileName%
echo 分析结果:
```

```
>>%fileName%
echo ----->>%fileName%
#依次打开目录下每一个
```

Monkey日志查询关键字并输出个数

```
(for %%a in (%str%)do (
  set n%%a=0&set/p= %%a : <nul>con
  for /f "delims=" %%b in ('findstr "%a" "%ff%")do (
    set h=%%b
    call :yky %%a)
  echo !n%%a!>con
  echo 关键字
```

%%a 共有

!n%%a! 处

```
)>>%fileName%
echo.>>%fileName%
#针对崩溃的日志输出其所在文件行数
```

echo 崩溃日志:

```
>>%fileName%
findstr "%str%" "%ff%">>%fileName%
echo/&pause&exit
:yky
set/a n%1+=1
set h=!h:*%1=!
if defined h if not "!h:*%1!="=="!h!" goto :yky
```

最终执行后，在脚本目录下会生成Result.txt文件记录异常出现的次数，如图4-11所示。

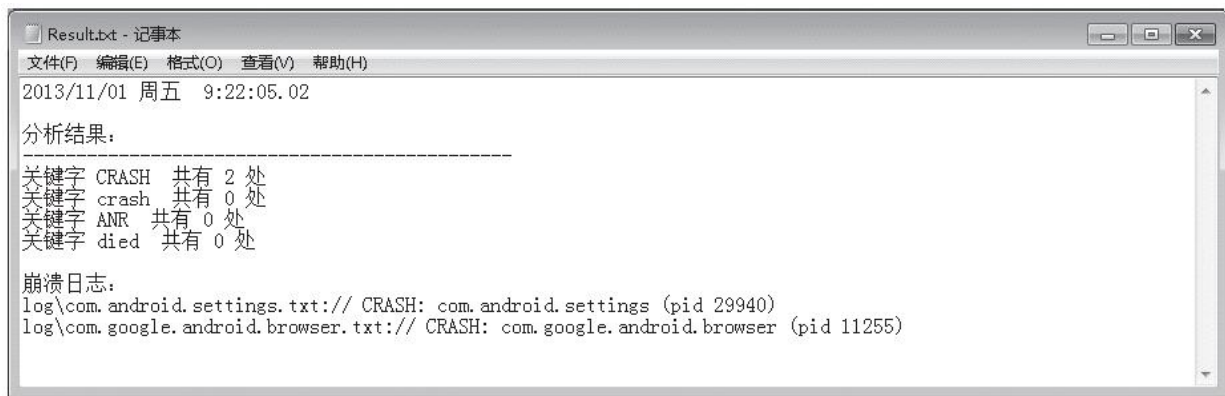


图4-11 日志分析结果文件

根据统计结果，可以得到Crash和ANR出现的次数，以及出现在哪个日志文件中，出现该错误的包名。如果需要更详细的错误信息，可以打开对应的Monkey日志文件查询。通过详细日志信息，测试可以定位到引起Crash的原因，以及出现Crash的代码行信息。这里给出常见的一些Crash错误信息，见表4-7。

表4-7 常见Crash信息表

Crash 关键字	Crash 原因
java.lang.NullPointerException	空指针异常
java.lang.ArrayIndexOutOfBoundsException	数组溢出
java.lang.ClassNotFoundException	类不存在
java.lang.ArithmeticException	数学运算异常
java.lang.IllegalArgumentException	方法参数异常
java.io.FileNotFoundException	文件未找到
java.lang.NumberFormatException	数值转化异常
java.lang.StackOverflowError	堆栈异常错误
java.lang.OutOfMemoryError	内存溢出错误

当获取到Crash和ANR日志信息后，理论上开发人员就可以开始根据日志内容分析和定位问题了。但事实上，要定位问题单靠日志信息还是非常困难的，有时候开发人员还需要知道问题复现的场景，同时增加更多的调试日志以协助定位。这时候，他们可能会期望测试能够复现问题或者提供问题出现场景和操作步骤。通常，测试人员可以通过使用同一个种子数（seed值），再次执行Monkey来尝试复现问题。这种方法比较费时，并且不是所有的随机Crash和ANR都可以通过这种方法来复现。那问题来了，在Monkey出现问题的时候有没有可能即时地截图并且记录下操作步骤呢？Monkey本身是没有这个能力的，但是通过一些Monkey改造可以实现该功能。在4.4节会详细介绍Monkey改造的方法，大家不妨带着这个问题继续往下看。

4.3 Monkey的基本原理

前面介绍了Monkey的各种参数以及其基本使用方法，顺带提到了在实践中Monkey页存在一定局限性，比如Monkey本身是不支持截图的，Monkey执行过程中网络断开后无法支持自动重连，等等。针对这些问题，可以通过改造Monkey源码的方式来实现。接下来这一节将重点介绍Monkey的代码框架及其内部实现逻辑，帮助大家更好地了解Monkey是怎样工作的，为接下来的4.4节Monkey源码改造打好基础。让我们先从Monkey的代码框架入手。

4.3.1 Monkey代码框架

前面说到了Monkey程序由一个名为“monkey”的shell脚本来启动执行，该脚本在Android文件系统中的存放路径是：/system/bin/monkey。这里简单解释一下此脚本，如下面的代码所示。

```
#将
base  设置为
system  路径

base=/system
#将
monkey.jar  路径设置为
CLASSPATH  环境遍历

export CLASSPATH=$base/framework/monkey.jar
trap "" HUP
#通过
app_process 命令启动

Monkey
exec app_process $base/bin com.android.commands.monkey.Monkey $*
```

通过脚本不难发现，该批处理通过app_process命令指向的是手机上framework目录下的一个monkey.jar包中的“com.android.commands.monkey.Monkey”类（这个类即为Monkey的入口函数所在类）。monkey.jar的源码位于Android源码的“\development\cmds\monkey\src\com\android\commands\monkey”目录下，如图4-12所示。

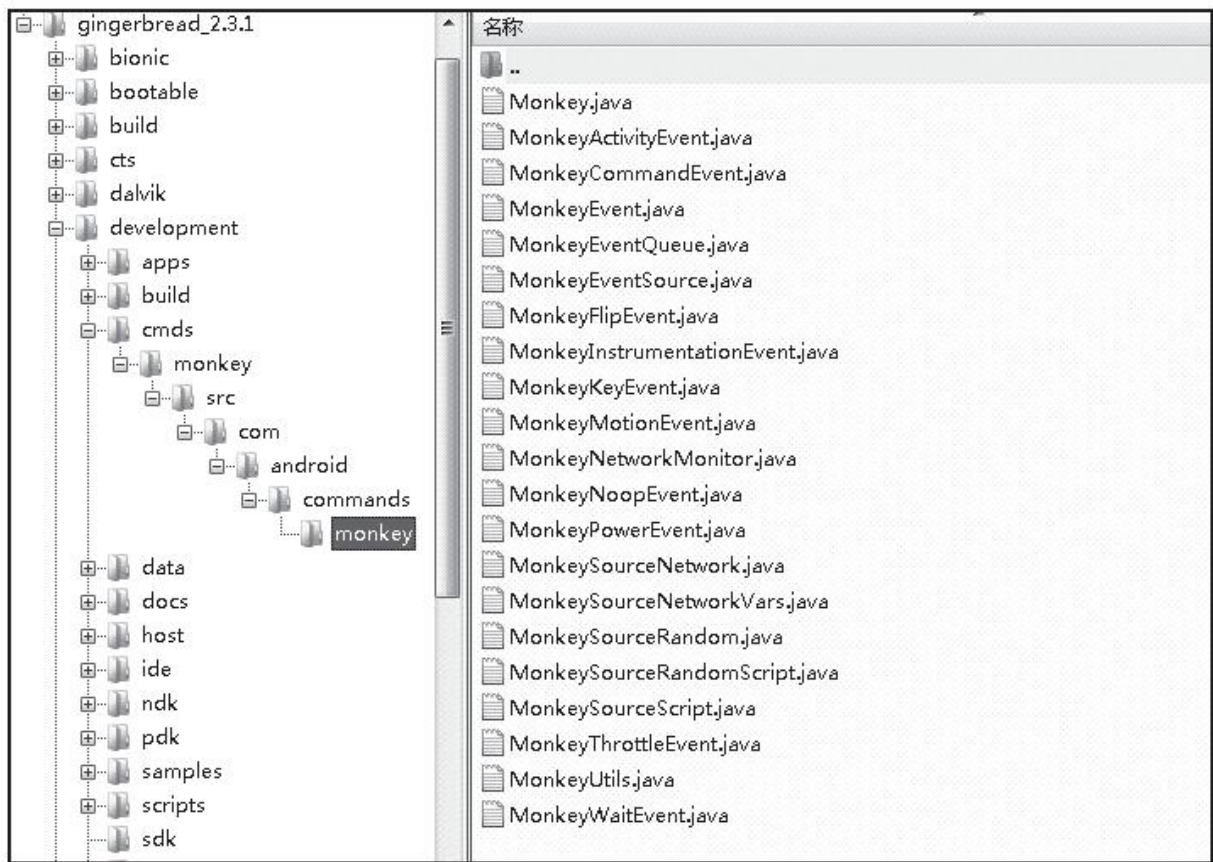


图4-12 Monkey源码文件列表

Monkey的代码框架如图4-13所示。

Monkey的代码核心模块主要包括主控、监控、事件源和事件四大部分：

- 主控模块：主控模块即**Monkey**类，是入口函数所在类，主要负责参数解析和赋值、初始化运行环境（导入**package**列表、检查内部配置、申请系统资源、初始化事件源、启动监控等）、执行**runMonkeyCycles（）**方法针对不同的事件源开始获取并执行不同的事件。

·**监控模块**：监控部分包括异常监控和网络监控两部分。异常监控通过**ActivityWatch**类来实现，主要监控**Activity**的**Crash**和**ANR**事件。网络监控通过**MonkeyNetworkMonitor**类来实现，主要用于统计运行期间移动网络和**Wi-Fi**网络的链接时长。

·**事件源模块**：事件源代表不同的事件来源。以**MonkeyEventSource**为基类，衍生出三个**Source**类，分别代表网络来源（如**Monkeyrunner**）、随机事件来源（常规**Monkey**命令）、脚本来源（通过**-f**参数指定的脚本）三种事件来源。事件源要做的事情有：从对应来源获取信息，并生成对应的事件，将其插入事件队列中。

·**事件模块**：事件代表了各种用户操作类型。以**MonkeyEvent**为基类，衍生出各种**Event**类，每一个**Event**类代表一种用户操作类型，如常见的点击、输入、滑动事件等。**MonkeyEvent**抽象类中提供了**injectEvent ()**方法，用于执行对应的事件。

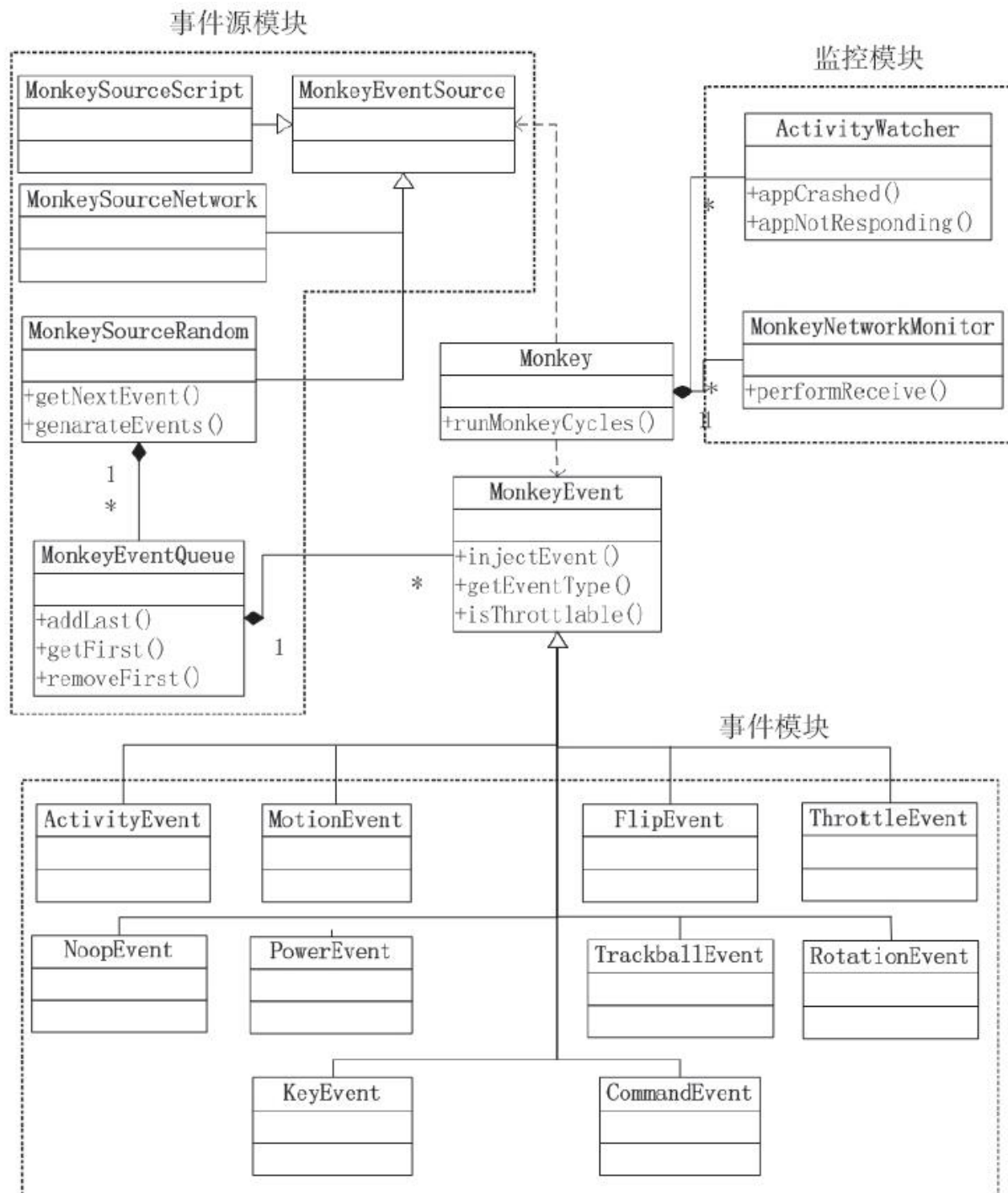


图4-13 Monkey的代码框架

4.3.2 Monkey代码逻辑详解

接下来看一下Monkey运行的流程。

首先，从入口函数入手，Monkey.java的main方法如代码清单4-9所示。

代码清单4-9 Monkey.java的main方法

```
public static void main(String[] args) {  
    #将进程名记入进程列表  
  
    Process.setArgV0("com.android.commands.monkey");  
    #运行  
  
    Monkey  
    int resultCode = (new Monkey()).run(args);  
    #退出  
  
    System.exit(resultCode);  
}
```

从代码中可以看到main方法是通过调用run方法来运行Monkey的。Monkey.java的run方法核心处理代码如代码清单4-10所示。

代码清单4-10 Monkey.java的run方法核心处理代码

```
//解析  
Monkey参数并逐一赋值  
  
if (!processOptions()) {  
    return -1;  
}
```

```

}...

//获取
ActivityManager、
WindowManager、
PackageManager三大系统资源

//通过
ActivityManager调用
setActivityController()方法对测试生命周期进行监控

//通过
WindowManager的方法来将事件注入到应用中，实现事件操作

//通过
PackageManager获取
Intent中应用列表，方便在多应用之间切换

if (!getSystemInterfaces()) {
    return -3;
}
...
//初始化事件源

//脚本事件源：带
-f参数且脚本数等于
1
if (mScriptFileNames != null && mScriptFileNames.size() == 1) {
    mEventSource = new MonkeySourceScript(mRandom, mScriptFileNames.get(0),
        mThrottle, mRandomizeThrottle, mProfileWaitTime, mDeviceSleepTime);
    mEventSource.setVerbose(mVerbose);
    mCountEvents = false;
} //脚本事件源

:带
-f参数且脚本数大于
1
} else if (mScriptFileNames != null && mScriptFileNames.size() > 1) {
    if (mSetupFileName != null) {
        mEventSource = new MonkeySourceRandomScript(mSetupFileName,
            mScriptFileNames, mThrottle, mRandomizeThrottle, mRandom,
            mProfileWaitTime, mDeviceSleepTime, mRandomizeScript);
        mCount++;
    }
}

```

```

    } else {
        mEventSource = new MonkeySourceRandomScript(mScriptFileNames,
            mThrottle, mRandomizeThrottle, mRandom,
            mProfileWaitTime, mDeviceSleepTime, mRandomizeScript);
    }
    mEventSource.setVerbose(mVerbose);
    mCountEvents = false;
//远程调用事件源: 带

--port参数

} else if (mServerPort != -1) {
    try {
        mEventSource = new MonkeySourceNetwork(mServerPort);
    } catch (IOException e) {
        System.out.println("Error binding to network socket.");
        return -5;
    }
    mCount = Integer.MAX_VALUE;
//随机事件源

} else {
    // random source by default
    if (mVerbose >= 2) { // check seeding performance
        System.out.println("// Seeded: " + mSeed);
    }
    mEventSource = new MonkeySourceRandom(mRandom, mMainApps, mThrottle,
mRandomizeThrottle);
    mEventSource.setVerbose(mVerbose);
    // set any of the factors that has been set
    for (int i = 0; i < MonkeySourceRandom.FACTORZ_COUNT; i++) {
        if (mFactors[i] <= 0.0f) {
            ((MonkeySourceRandom) mEventSource).setFactors(i, mFactors[i]);
        }
    }
    // in random mode, we start with a random activity
    ((MonkeySourceRandom) mEventSource).generateActivity();
}...

//启动网络监控程序

mNetworkMonitor.start();
//开始执行

Monkey
int crashedAtCycle = runMonkeyCycles();
mNetworkMonitor.stop();
//打印

ANR、

Crash等报告

synchronized (this) {
    ...

```

}

从代码解析可以看到，run方法主要做了以下几件事。

- 参数解析和参数赋值。
- 申请系统资源。
- 初始化事件源。
- 启动网络监控程序。
- 调用runMonkeyCycles方法执行Monkey。

再来看一下runMonkeyCycles方法又做了什么。runMonkeyCycles中最核心的逻辑如代码清单4-11所示。

代码清单4-11 runMonkeyCycles方法核心代码

```
while (!systemCrashed && cycleCounter < mCount) {
    ...

    //通过不同事件源读取下一事件

getNextEvent()
    MonkeyEvent ev = mEventSource.getNextEvent();
    if (ev != null) {
        //通过

injectEvent()将事件注入系统中

        int injectCode = ev.injectEvent(mWm, mAm, mVerbose);
        ...
    }
}
```

```
    }  
}
```

从代码分析可知，runMonkeyCycles做了两件事，一是调用了事件源的getNextEvent方法读取下一个事件，二是调用事件的injectEvent方法执行该事件。

针对不同的事件源，getNextEvent方法的具体实现是不同的，不过它们的目的都只有一个，即获取下一个事件。以随机事件源（monkeySourceRandom）为例，来看一下它是怎么实现getNextEvent的，如代码清单4-12所示。

代码清单4-12 getNextEvent方法核心代码

```
public MonkeyEvent getNextEvent() {  
    //mQ表示事件队列  
  
    //在  
    monkeySourceRandom初始化时会创建一个事件队列用于存放被执行事件  
  
    //假如当前事件队列为空，则创建事件并加入队列中  
  
    if (mQ.isEmpty()) {  
        generateEvents();  
    }  
    //获取当前队列中的第一个事件  
  
    mEventCount++;  
    MonkeyEvent e = mQ.getFirst();  
    //从队列中删除被取出的事件  
  
    mQ.removeFirst();  
    return e;  
}
```

monkeySourceRandom在初始化的时候会创建一个事件队列mQ用于存放Monkey事件。当外部调用getNextEvent方法时，会先去判断mQ队列中是否有事件，如果队列中没有事件，则调用generateEvents方法，根据事件源自身的规则生成事件流，并将其存放到mQ队列中；如果队列中有事件，则会取出第一个事件返回，并同时从队列中删除该事件。

而获取事件后需要做的就是执行相应的事件，也就是将事件通过injectEvent方法逐一注入系统中。几乎每个Event事件都有对injectEvent方法的具体实现。这里以触摸事件（MonkeyTouchEvent）为例来看一下它是如何进行事件注入的，如代码清单4-13所示。

代码清单4-13 MonkeyTouchEvent代码注入实现

```
public int injectEvent(IWindowManager iwm, IActivityManager iam, int verbose) {
    String note;
    //输出事件执行日志

    if ((verbose > 0 && !mIntermediateNote) || verbose > 1) {
        if (mAction == MotionEvent.ACTION_DOWN) {
            note = "DOWN";
        } else if (mAction == MotionEvent.ACTION_UP) {
            note = "UP";
        } else {
            note = "MOVE";
        }
        System.out.println(":Sending Pointer ACTION_" + note +
            " x=" + mX + " y=" + mY);
    }
    try {
        //获取当前事件及其类型

        int type = this.getEventType();
        MotionEvent me = getEvent();
        //调用
```

WindowManager的方法实现事件注入

```
        //如果是
        Point类型, 调用
        injectPointerEvent接口

        //如果是
        Trackball类型, 调用
        injectTrackballEvent接口

        if ((type == MonkeyEvent.EVENT_TYPE_POINTER &&
            !iwm.injectPointerEvent(me, false))
            || (type == MonkeyEvent.EVENT_TYPE_TRACKBALL &&
            !iwm.injectTrackballEvent(me, false))) {
            return MonkeyEvent.INJECT_FAIL;
        }
    } catch (RemoteException ex) {
        return MonkeyEvent.INJECT_ERROR_REMOTE_EXCEPTION;
    }
    return MonkeyEvent.INJECT_SUCCESS;
}
```

这里, 执行事件通过调用WindowManager的接口来实现事件注入。

最后来回顾一下整个Monkey事件执行的流程。

- (1) 初始化事件源, 创建事件队列。
- (2) 通过getNextEvent () 方法循环获取事件。
- (3) 通过injectEvent方法调用WindowManager的方法将事件注入系统中。

4.4 Monkey扩展应用示例

前面讲到了Monkey的基本使用方法及其原理，可以看到Monkey功能还是很强大的，它不但可以进行随机测试，还可以执行指定脚本，结合adb命令还可以监控各种性能数据。但它毕竟是一款为稳定性测试而准备的小工具，所以存在很多局限性，正如前面Monkey实践一节中提到的，Monkey不提供截屏功能，因此测试很难找到问题复现的场景；Monkey无法进行控件识别，对事件流控制能力很微弱；执行过程中容易误点工具栏导致Wi-Fi关闭，影响测试效果；等等。本节重点介绍的就是如何通过Monkey源码改造的方法来解决上述问题，以更好地提升Monkey的使用效果。



注意 随书提供的源码是Windows系统下可运行的Monkey工程，仅适用于Android 4.1.2版本的手机。

要改造Monkey，就要先了解如何重编译Monkey。

4.4.1 Monkey代码重编译执行方法

Monkey重编译的方法有两种，一种是在Linux环境下编译，另一种是在Windows环境下编译。因为在Windows环境下编译更为常见，所以这里会重点介绍第二种方法。

1.Linux环境下编译

在Linux环境下，下载要测试Android系统版本对应的全部源代码，进入源码目录。

步骤1：执行`.build/envsetup.sh`，设置Android的编译环境。

步骤2：执行`make monkey`开始编译Monkey。

编译成功后，可在`/out/target/product/generic/system/framework/`中获取Monkey.jar包。

2.Windows环境下编译

Windows环境下的编译要稍微复杂一点。

步骤1：创建Monkey项目。同样也是需要下载要测试Android系统版本对应的全部源代码，在`/development/cmds/monkey`目录下找到

Monkey的工程源码。在Eclipse中新建一个Java工程，把Monkey源码导入进去。

步骤2: 设置Java Build Path。选中对应项目，在顶部菜单栏依次选题Project → Properties → Java Build Path → Libraries，添加两个jar文件：android.jar和framework.jar。其中android.jar可以从Android Sdk中platforms\android-X\目录下获取；framework.jar可以通过以下两种方式获取。

(1) (推荐) 从在Linux环境下Android源码根目录执行make update-api编译生成，如截图中的classes_dex2jar.jar文件就是通过Android源码编译生成的。

(2) 直接从Android手机上/system/framework目录下获取已经编译好的framework.jar文件，把这个framework.jar解压，取出其dex，然后把它的dex通过dex2jar工具转换为jar包，导入工程。

添加android和framework的jar包后，还需要将framework的jar包顺序调整到顶部，如图4-14所示。

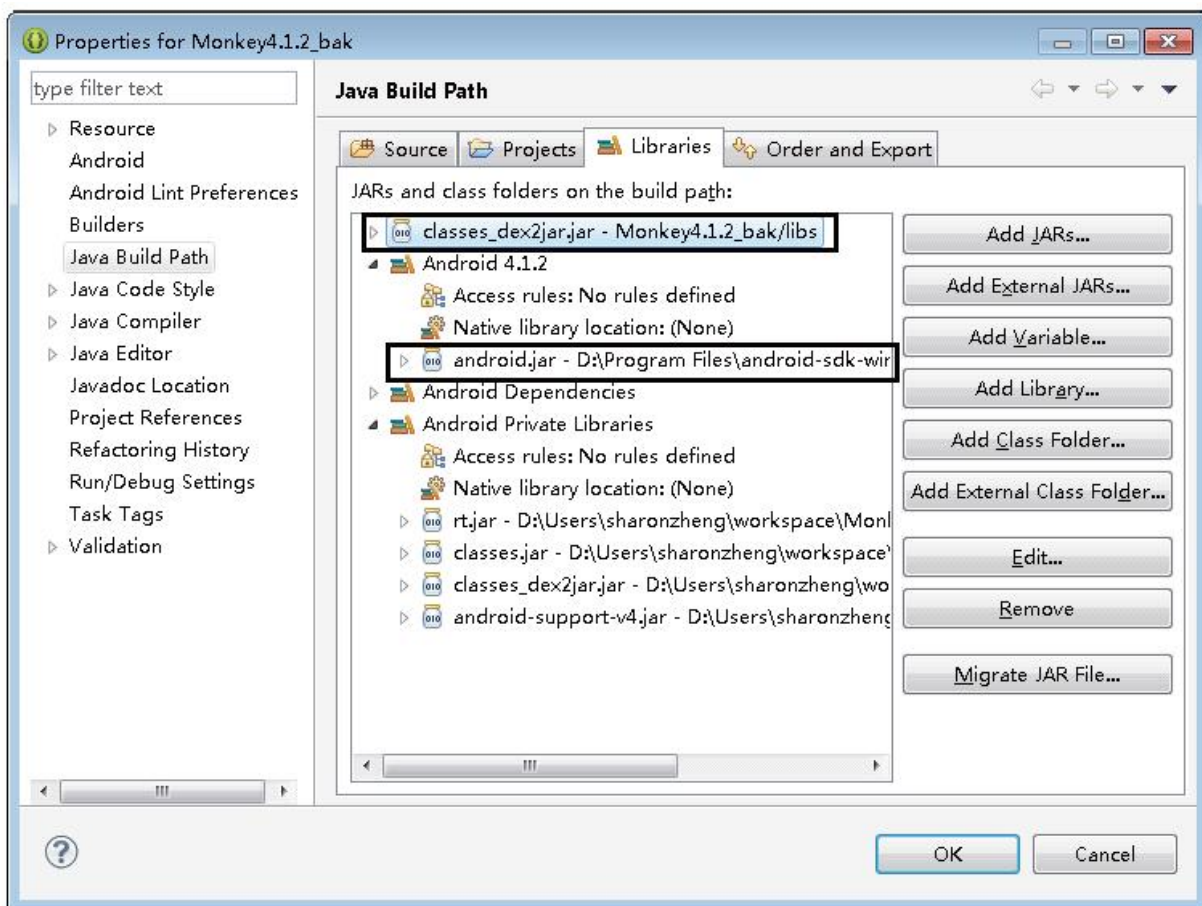


图4-14 添加android和framework的jar包

步骤3: 编译生成jar包。选择Monkey项目，单击右键→单击Export→选择输出的Jar包类型为“JAR file”类，单击“Next”按钮，如图4-15所示。

选择对应的构建工程，填写jar包输出路径，单击“Next”按钮，如图4-16所示。

进入打包选项页面，这里用默认选项即可，直接单击“Next”按钮，如图4-17所示。

选择工程中main函数所在的类，单击“Finish”按钮，如图4-18所示。

编译完成后，在指定目录下就会生成对应的Monkey.jar包了。

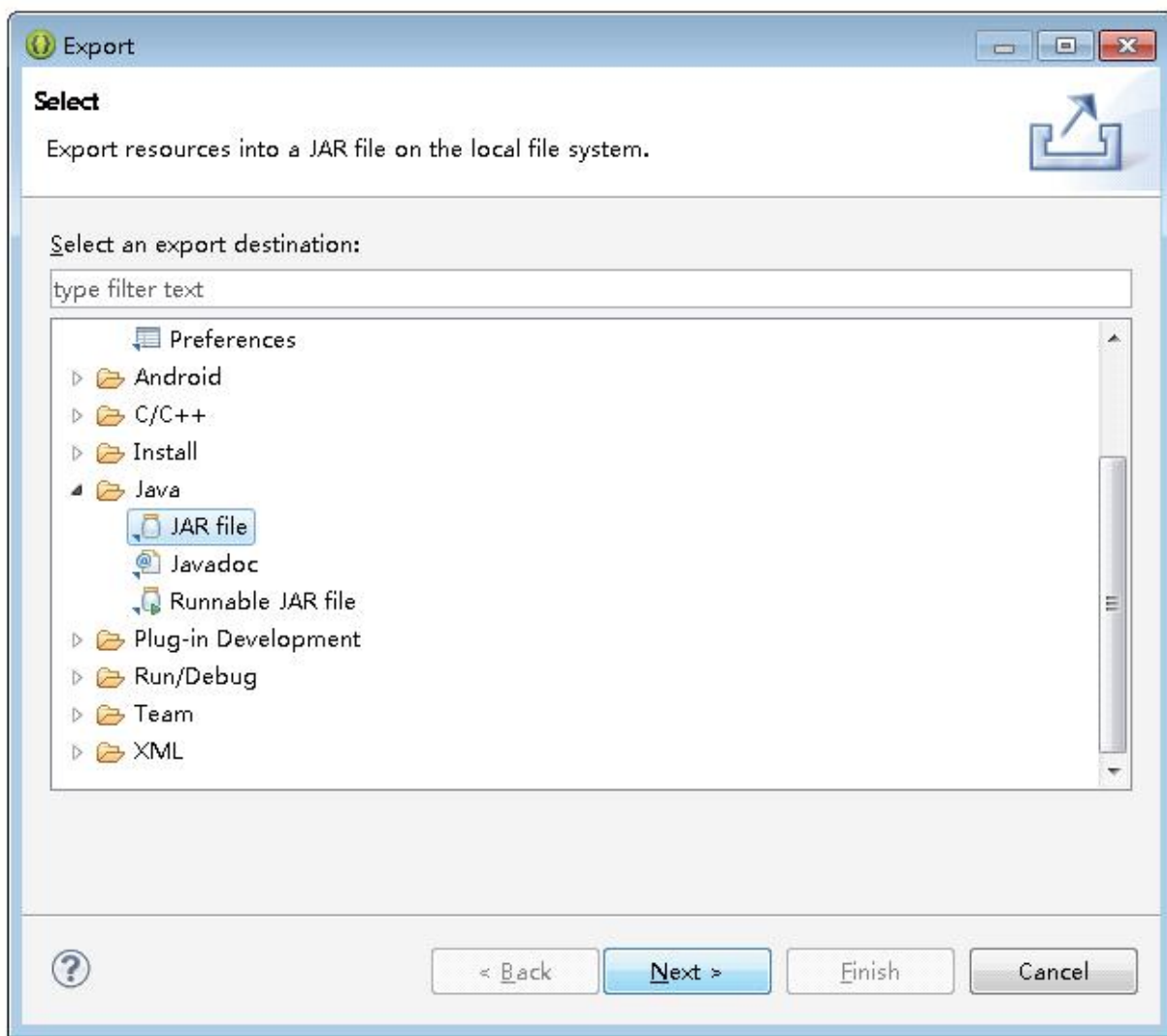


图4-15 选择JAR file

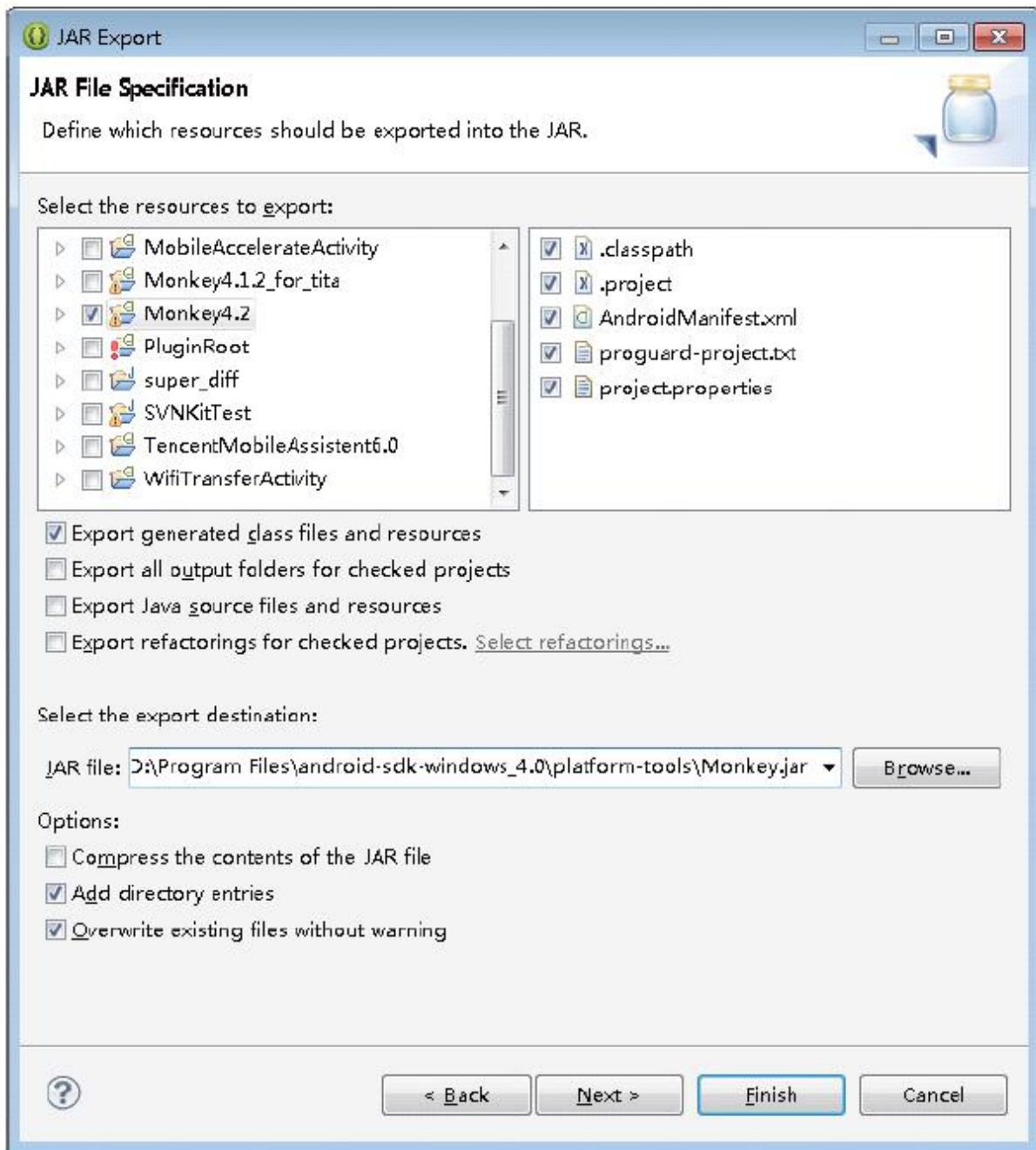


图4-16 填写jar包输出路径

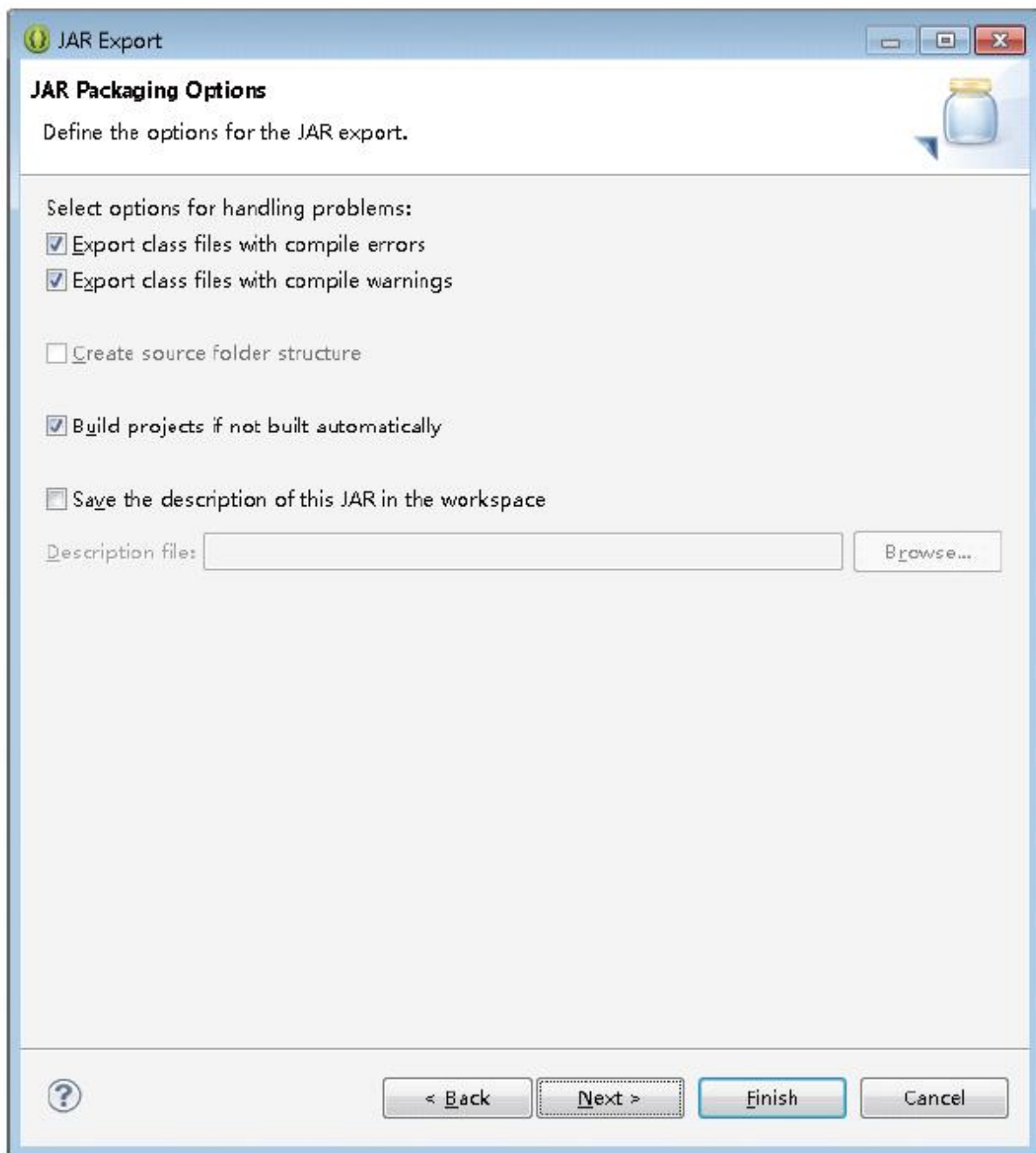


图4-17 打包选项页面

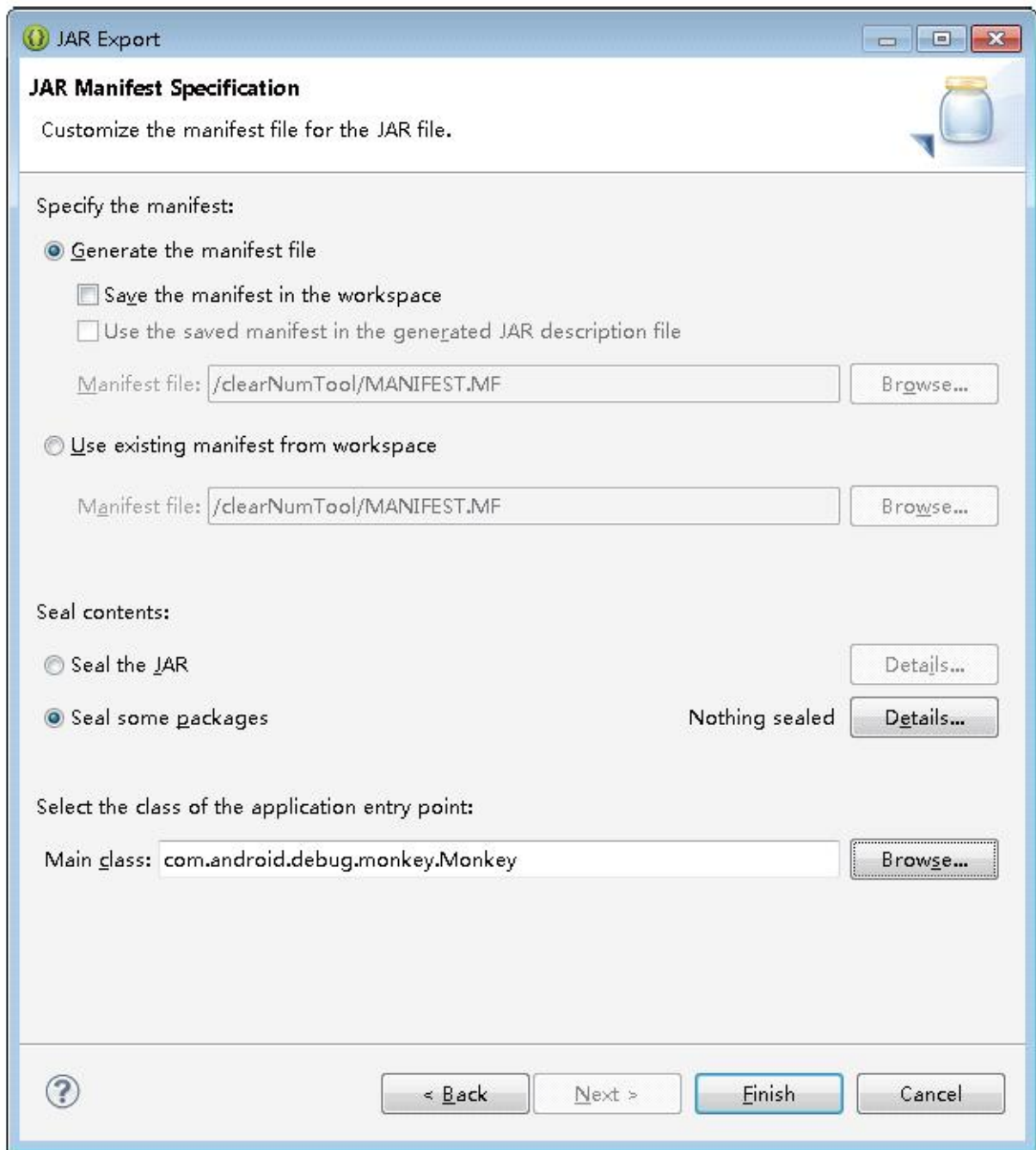


图4-18 选择main函数所在的类

步骤4: 转换Monkey.jar包。Eclipse编译出来的jar包是不能直接放到Android手机上运行的，在Android上无法像Java中那样方便地动态加

载jar。原因是：Android的虚拟机（Dalvik VM）是不能识别Java打出的jar的byte code的，这里需要通过Android sdk中的dx工具来优化转换成Dalvik byte code才行。

将打包好的jar复制到SDK安装目录android-sdk-windows\platform-tools下，打开命令行进入platform-tools目录，执行命令：

```
dx --dex --output=<生成的目标文件>  
> <要转换的文件>  
>
```

执行方法如图4-19所示



图4-19 转化Monkey.jar包执行方法

此时在android-sdk-windows\platform-tools目录下会生成最终的Monkey可执行包，这个jar包可直接在手机上运行。这里输出的是Monkeytest.jar。

3.重编译的包运行方法

假设前面重编译后生成的Monkey文件为Monkey.jar文件，测试如何在手机上执行这个新文件呢？

要运行重编译后的Monkey.jar有以下两个前提条件。

- 手机拥有root权限。

- 手机Android版本与Monkey.jar包的Android版本一致。

（由于不同版本的Android系统API不同，因此不同版本的Monkey包也是不能通用的。例如：Android 4.2版本的Monkey只能在Android 4.2的系统上运行。）

步骤1：创建启动shell脚本。

在本地新建一个用于启动Monkey的shell脚本，输入以下命令，并保存成Monkey。这个文件是用来启动和执行Monkey.jar的，如下面的代码所示。

```
# Script to start "monkeytest" on the device, which has a very rudimentary
# shell.
#这里要填写编译后生成的

jar文件名称

export CLASSPATH=/data/ Monkey.jar
#这里要填写

jar文件中的入口函数所在类

exec app_process /data com.android.debug.monkey.Monkey $*
```

步骤2：上传脚本和jar包到手机。

将步骤1创建的Monkey脚本和Monkey.jar包上传到手机的/data/local目录（可自己定义，与shell脚本中的目录一致即可），并将Monkey文件修改成可执行权限，如下面代码所示。

```
adb push Monkey.jar /data
adb push monkey /data
adb shell chmod777 /data/monkey
```

个别手机上执行chmod命令时会报Segmentation Fault错误，这时可以先adb shell进入，通过su root命令切换到root下，再执行chmod 777/data/monkey即可。

步骤3: 运行monkey。

通过命令行窗口，输入adb shell./data/local/monkey<option>[count]命令启动Monkey.jar包即可运行Monkey。

4.4.2 Monkey截图优化

掌握重编译Monkey的方法后，接下来要开始进行Monkey源码改造了。第一个改造就是截图改造。Monkey使用过程中最大的难题就是如何获取异常出现的场景。虽然Monkey在执行过程中提供了日志来记录事件执行顺序，但是光靠日志来定位异常出现的场景并复现它是非常困难的。当Monkey执行过程中出现异常时，若可以对应进行截图并记录异常出现前执行的操作，就可以清晰地知道异常出现的场景，也便于定位和解决问题。

具体改造方法如下：

测试期望实现的是在每个事件执行过程中增加截图并在图片上画出事件轨迹。这里以屏幕触摸操作为例，首先找到触摸事件所在的文件MonkeyMotionEvent.java，找到负责执行该事件的injectMotionEvent方法。先来看一下它的实现，如代码清单4-14所示。

代码清单4-14 injectMonkeyEvent方法源代码

```
public int injectEvent(IWindowManager iwm, IActivityManager iam, int verbose) {
    String note;
    if ((verbose > 0 && !mIntermediateNote) || verbose > 1) {
        if (mAction == MotionEvent.ACTION_DOWN) {
            note = "DOWN";
        } else if (mAction == MotionEvent.ACTION_UP) {
            note = "UP";
        } else {
            note = "MOVE";
        }
    }
}
```

```

        System.out.println(":Sending Pointer ACTION_" + note +
            " x=" + mX + " y=" + mY);
    }
    try {
        int type = this.getEventType();
        MotionEvent me = getEvent();
        if ((type == MonkeyEvent.EVENT_TYPE_POINTER &&
            !iwm.injectPointerEvent(me, false))
            || (type == MonkeyEvent.EVENT_TYPE_TRACKBALL &&
            !iwm.injectTrackballEvent(me, false))) {
            return MonkeyEvent.INJECT_FAIL;
        }
    } catch (RemoteException ex) {
        return MonkeyEvent.INJECT_ERROR_REMOTE_EXCEPTION;
    }
    return MonkeyEvent.INJECT_SUCCESS;
}

```

这里是通过WindowManager的一个实例访问InjectPointerEvent和injectTrackballEvent接口来实现事件注入的。其中me参数表示要执行的一个MotionEvent事件。常规的触摸操作是由三类MotionEvent事件组成的，一类是ACTION_DOWN类型的MotionEvent，表示按下；一类是ACTION_MOVE类型的MotionEvent，表示移动；一类是ACTION_UP事件，表示抬起。ACTION_DOWN是每个点击事件的开始，只要在这个地方添加一个截图方法，即可对点击事件进行截图。

除了截图以外，还需要记录事件轨迹。因此在每个事件中，需要把点坐标记录下来，存放到一个PointList的点队列中。当出现ACTION_UP时，意味着整个点击操作已经完成了，此时把最后一个点坐标加到队列后，再把队列中保存的所有点坐标按一定的规则画到开始时截的图中，压缩图片并保存，此时就完成了点击操作的截图和记录。最后别忘了清空PointList队列。

修改后的injectEvent方法如代码清单4-15所示。

代码清单4-15 在injectEvent方法中增加截图和画轨迹

```
public int injectEvent(IWindowManager iwm, IActivityManager iam, int verbose) {
    String note;
    if ((verbose > 0 && !mIntermediateNote) || verbose > 1) {
        if (mAction == MotionEvent.ACTION_DOWN) {
            note = "DOWN";
            //截图

            ImageUtils.TakeScreenshot();
            //获取当前点击坐标, 存到队列中

            ImageUtils.addPoint(me.getX(),me.getY());
        } else if (mAction == MotionEvent.ACTION_UP) {
            note = "UP";
            //获取当前点击坐标, 存到队列中

            ImageUtils.addPoint(me.getX(),me.getY());
            //把队列中的点击坐标画到图片上

            Bitmap bc=ImageUtils.drawPoint(ImageUtils.scaleBitmap());
            ImageUtils.removePointList();//清空队列

        } else {
            note = "MOVE";
            //获取当前点击坐标, 存到队列中

            ImageUtils.addPoint(me.getX(),me.getY());
        }
        System.out.println(":Sending Pointer ACTION_" + note +
            " x=" + mX + " y=" + mY);
    }
    ...
}
```

上面用到的截图方法TakeScreenshot是直接调用一个第三方的截屏工具gsnapCap进行截屏的，具体实现如代码清单4-16所示。

代码清单4-16 截图方法的实现

```

public static void TakeScreenshot() {
    //以时间戳命名截图文件

    String filename=String.valueOf(System.currentTimeMillis());
    try {
        //通过命令行调用了

gsnapCap工具进行截图

        Process p = Runtime.getRuntime().exec("/data/local/gsnapCap
            "+filepath+filename+".jpg /dev/graphics/fb0 1");
        p.waitFor();
        //异常处理

    } catch (IOException e) {
        System.out.println("cannot write picture,please check your gsnapCap");
    } catch (InterruptedException e) {
        System.out.println("cannot write picture,please check your gsnapCap");
    }
}

```

前面在injectEvent方法中调用了一个drawPoint方法将操作轨迹画到图片上，drawPoint方法具体实现如代码清单4-17所示。

代码清单4-17 画操作轨迹方法的实现

```

public static Bitmap drawPoint(Bitmap src) {
    //判断传入的

    Bitmap图片文件是否为空

    if (src == null) {
        return null;
    }
    int w = src.getWidth();
    int h = src.getHeight();
    //创建一个新的和

    SRC长度宽度一样的位图

    Bitmap newb= Bitmap.createBitmap(w, h, Config.ARGB_8888);
    Canvas cv = new Canvas(newb);
    cv.drawBitmap(src, 0, 0, null);
    MyCanvas mycanvas=new MyCanvas();
    mycanvas.setPaintDefaultStyle();

```

```

mycanvas.onDraw(cv);
//获取起始位置和结束位置的点

Point to=(Point) PointList.get(PointList.size()-1);
Point from=(Point) PointList.get(0);
//只有两个点，直接在图片上画点及点击坐标

if(PointList.size()<3){
    mycanvas.drawText("⊙", (int)to.getX()-10, (int)to.getY()+10);
    mycanvas.drawText("Click ("+(int)to.getX()+", "+(int)to.getY()+")",
        (int)to.getX()-100, (int)to.getY()-15);
    //有多个点，则画箭头并标注操作坐标

    //为避免文字超出图片范围，所以要根据起始点和终点的位置来调整文字展示位置

    }else if((int)from.getY()<(int)to.getY()&&(int)from.getX()>w/2){
        mycanvas.drawAL((int)from.getX(), (int)from.getY(), (int)to.getX(),
        (int)to.getY());;
        mycanvas.drawText("From ("+(int)from.getX()+", "+(int)from.getY()+")",
        (int)from.getX()-100, (int)from.getY()-15);
        mycanvas.drawText("To ("+(int)to.getX()+", "+(int)to.getY()+")",
        (int)to.getX()-100, (int)to.getY()+30);
    }else if((int)from.getY()>(int)to.getY()&&(int)from.getX()>w/2){
        mycanvas.drawAL((int)from.getX(), (int)from.getY(), (int)to.getX(),
        (int)to.getY());;
        mycanvas.drawText("From ("+(int)from.getX()+", "+(int)from.getY()+")",
        (int)from.getX()-100, (int)from.getY()+30);
        mycanvas.drawText("To ("+(int)to.getX()+", "+(int)to.getY()+")",
        (int)to.getX()-100, (int)to.getY()-15);
    }else if((int)from.getY()<(int)to.getY()&&(int)from.getX()<=w/2){
        mycanvas.drawAL((int)from.getX(), (int)from.getY(), (int)to.getX(),
        (int)to.getY());;
        mycanvas.drawText("From ("+(int)from.getX()+", "+(int)from.getY()+")",
        (int)from.getX()-50, (int)from.getY()-15);
        mycanvas.drawText("To ("+(int)to.getX()+", "+(int)to.getY()+")",
        (int)to.getX()-50, (int)to.getY()+30);
    }else if((int)from.getY()>(int)to.getY()&&(int)from.getX()<=w/2){
        mycanvas.drawAL((int)from.getX(), (int)from.getY(), (int)to.getX(),
        (int)to.getY());;
        mycanvas.drawText("From ("+(int)from.getX()+", "+(int)from.getY()+")",
        (int)from.getX()-50, (int)from.getY()+30);
        mycanvas.drawText("To ("+(int)to.getX()+", "+(int)to.getY()+")",
        (int)to.getX()-50, (int)to.getY()-15);
    }
    //保存，描点完成

    cv=mycanvas.getCanvas();
    cv.save(Canvas.ALL_SAVE_FLAG);
    cv.restore();
    return newb;
}

```

参照上面的修改思路，将Monkey的其他方法也进行类似的修改。这样，Monkey每执行一个操作，系统就会自动对其进行截图描点了。

由于手机SD卡空间有限，如果不断地往手机里添加文件，SD卡很快就会被填满。因此还需要增加一个定时清理的操作，即只保存最新的30张图片，一旦图片超过30张，就会把最早的一张图片删除。删除的实现方法如代码清单4-18所示，只需要在每次创建图片时，调用这个方法即可。

代码清单4-18 删除SD卡文件方法的实现

```
public static void deletefile(){
    File files = new File(filepath);
    File filelist[] = files.listFiles();
    File file =null;
    long time=0;
    if(filelist==null){
        return;
    }
    if(filelist.length>30){
        for (int i=0;i<filelist.length;i++){
            String tempname=filelist[i].getName();
            tempname=tempname.substring(0,tempname.lastIndexOf('.'));
            long temptime= Long.valueOf(tempname);
            if(temptime<time||time==0){
                time=temptime;
                file=filelist[i];
            }
        }
        file.delete();
    }
}
```

前面做了这么多，最终当然是希望被测程序在跑Monkey出现Crash、ANR时，把截图保存下来。因此需要在程序中加入下面这几行代码，当出现Crash等异常时，会将最新的30张截图复制下来，存放到

一个以“Crash+当前时间差”命名的文件夹中，具体实现如代码清单4-19所示。

代码清单4-19 Crash和ANR时保持截图的实现

```
private void reportAnrTraces() {
    try {
        Thread.sleep(5 * 1000);
    } catch (InterruptedException e) {
    }
    commandLineReport("anr traces", "cat /data/anr/traces.txt");
    //把截图复制过去

    ImageUtils.TakeScreenshot();

    FileOperate.copyFolder(StorageDirectory+"/Monkey/Pic",StorageDirectory+"/Monkey/ANR/ANR"+String.valueOf(System.currentTimeMillis()));
}
```

这样Monkey截图功能就完成了，来看一下执行的效果。当手机执行完Monkey后，假如执行过程中出现了Crash或ANR，那么在sdcard的Monkey目录下会对应生成Crash和ANR目录，并保存发送异常之前的30张屏幕截图，如图4-20所示。

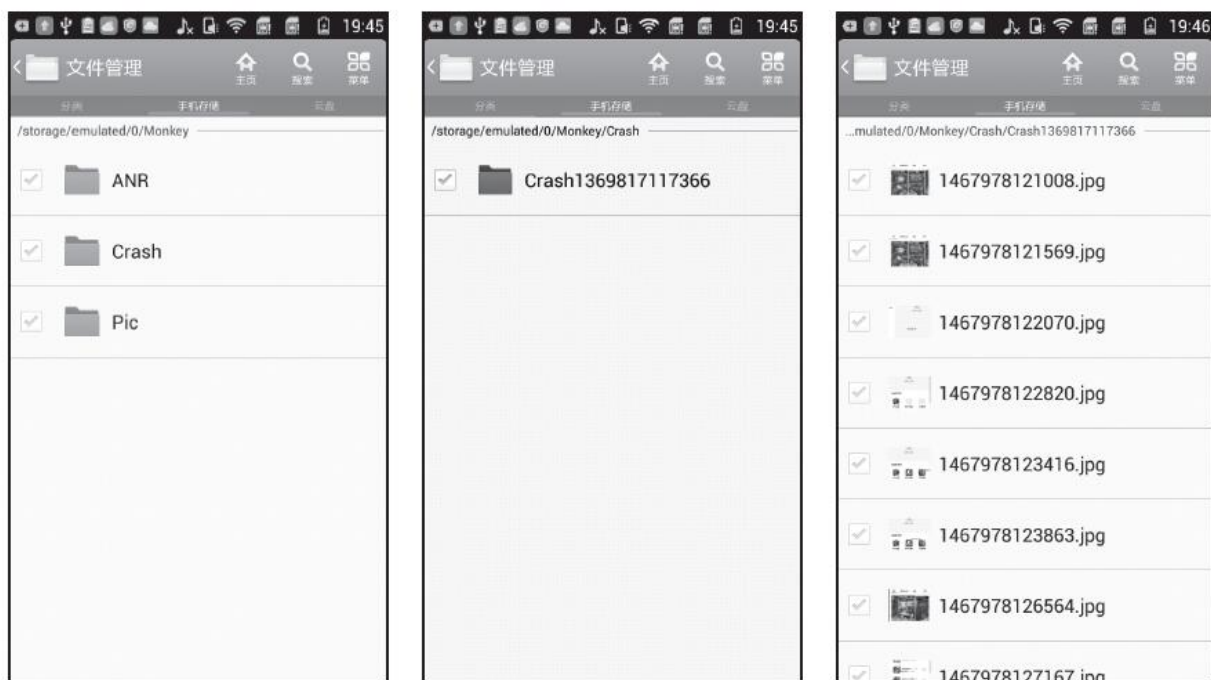


图4-20 Crash和ANR目录中保存的截图

图片上面会显示当前事件的操作坐标，如图4-21所示。



图4-21 Monkey截图

4.4.3 Monkey Wi-Fi自动重连优化

我们知道大部分的应用程序是需要联网的，假如Monkey在执行过程中Wi-Fi断开了怎么办？由于Monkey执行的是随机事件流，过程中的操作无法控制，用户很容易误点到工具栏而导致Wi-Fi断开。对于需要联网的应用，当Wi-Fi断开后，很多页面都会无法打开，此时Monkey执行的效果会相当不理想。相信这也是绝大多数用户遇到的问题，当前小节介绍的就是如何通过Monkey改造来实现Wi-Fi断开重连的功能。

首先，新增一个用于Wi-Fi监控的事件MonkeyWifiEvent。在Monkey中新增一类事件有以下两个步骤。

(1) 在MonkeyEvent新增一个eventType类型，如代码清单4-20所示。

代码清单4-20 Wi-Fi重连的EventType

```
public abstract class MonkeyEvent {
    protected int eventType;
    public static final int EVENT_TYPE_KEY = 0;
    public static final int EVENT_TYPE_TOUCH = 1;
    public static final int EVENT_TYPE_TRACKBALL = 2;
    public static final int EVENT_TYPE_ROTATION = 3; // Screen rotation
    public static final int EVENT_TYPE_ACTIVITY = 4;
    public static final int EVENT_TYPE_FLIP = 5;
    public static final int EVENT_TYPE_THROTTLE = 6;
    public static final int EVENT_TYPE_NOOP = 7;
    #新增一个
```

Wi-Fi监控的事件类型

```
public static final int EVENT_TYPE_WifiCheck = 9;...
```

(2) 新增对应事件的MonkeyWifiEvent类，需继承自MonkeyEvent类，如代码清单4-21所示。

代码清单4-21 Wi-Fi重连的MonkeyEvent类实现

```
package com.android.debug.monkey;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import android.app.IActivityManager;
import android.view.IWindowManager;
import com.android.debug.manager.wifiManager;
public class MonkeyWifiEvent extends MonkeyEvent{
    // 初始方法

    public MonkeyWifiEvent() {
        super(MonkeyEvent.EVENT_TYPE_WifiCheck);
    }
    // 调用

    CheckWifiConnection()方法检查

    Wi-Fi连接

    public int injectEvent(IWindowManager iwm, IActivityManager iam,int verbose){
        System.out.println("Check Wifi Conection.");
        wifiManager.CheckWifiConnection();
        return MonkeyEvent.INJECT_SUCCESS;
    }
}
```

从上面的代码可以看到，该事件是通过调用CheckWifiConnection
() 方法来检查Wi-Fi连接并自动重连的。

CheckWifiConnection () 方法的实现很简单，首先初始化一个WifiManager的对象，调用其getWifiEnabledState方法，检查当前Wi-Fi是否连接，当判断为Wi-Fi无连接时，调用setWifiEnabled方法打开Wi-Fi。等待Wi-Fi打开后，通过getConfiguredNetworks方法获取Wi-Fi列表，并遍历列表查找需要连接的Wi-Fi的SSID。查找到后，连接到对应的Wi-Fi上。具体实现如代码清单4-22所示。

代码清单4-22 Wi-Fi重连方法具体实现

```
public static void CheckWifiConnection(){
    IWifiManager im=IWifiManager.Stub.asInterface(ServiceManager
        .getService("wifi"));
    try {
        int state=im.getWifiEnabledState();
        System.out.println(state);
        WifiInfo wi=im.getConnectionInfo();
        if(state!=3){
            //打开

Wi-Fi
            System.out.println("Wifi not conect, connecting wifi.");
            im.setWifiEnabled(true);
            //等待

Wi-Fi打开, 然后连接

freewifi
            for(int i=0;i<90;i++){
                if(im.getWifiEnabledState()==3){
                    //连接

freewifi
                    List<WifiConfiguration> t=im.getConfiguredNetworks();
                    if(t!=null){
                        for(int j=0;j<t.size();j++){
                            if(t.get(j).SSID.indexOf("Tencent-FreeWiFi")!=-1){
                                int networkid=t.get(j).networkId;
                                im.enableNetwork(networkid, true);
                                Thread.sleep(7000);
                            }
                        }
                        break;
                    }
                }
            }
            break;
        }else{
            hread.sleep(2000);
        }
    }
}
```

```

        }
    }
}
} catch (RemoteException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (SecurityException e) {
    e.printStackTrace();
}
}
}

```

前面说的需求是实现定时监控，所以需要在Monkey.java中的runMonkeyCycles () 下每隔1000个事件就插入一个Wi-Fi监控事件，实现如代码清单4-23所示。

代码清单4-23 runMonkeyCycles方法中增加Wi-Fi监控事件

```

private int runMonkeyCycles() {
    int eventCounter = 0;
    int cycleCounter = 0;
    boolean shouldReportAnrTraces = false;
    boolean shouldReportDumpsysMemInfo = false;
    boolean shouldAbort = false;
    boolean systemCrashed = false;
    // TO DO : The count should apply to each of the script file.
    while (!systemCrashed && cycleCounter < mCount) {
        ...
        //添加

Wi-Fi检查的事件-

sharon
        if(cycleCounter%1000==0){
            try {
                addWifiEvent();
            } catch (RemoteException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
    System.out.println("Events injected: " + eventCounter);
    return eventCounter;
}

```

这样，当Monkey每执行完1000个事件后，就会去检测一下Wi-Fi的连接状态，当发现Wi-Fi断开就会自动重连。重新编译一下Monkey，然后看一下效果，当Monkey检查到Wi-Fi断开如图4-22所示时，会自动重连，如图4-23所示。



图4-22 执行Monkey时网络被断开



图4-23 Monkey自动重连网络

4.4.4 Monkey扩展应用的优点和缺点

前面介绍了几个通过源码改造扩展使用Monkey的案例，这里来总结一下Monkey改造的优点和缺点。Monkey改造的优点非常明显：

- 不依赖PC机，是一种很好的单机自动化模式。
- 在Monkey内部可以调用系统底层接口，做到很多App做不到的事情。
- 基于Monkey源码的改造可以做很多个性化定制。

其缺点是：

- 被测手机最好是root手机。

如前面所说，在执行Monkey的过程中需要对Monkey的jar包和shell脚本进行授权，如果不是root手机，这一步会非常麻烦。

- 不同Android版本的Monkey不通用，不利于做版本适配。

因为Monkey是Android系统自带的，不同Android版本的Monkey的代码会存在差异，例如2.3版本的Monkey执行触摸方法时是通过调用IWindowManager的injectPointerEvent方法来实现的。

```
WindowManager.getInstance().injectPointerEvent(me, false)
```

而4.0版本的Monkey则是通过调用InputManager的injectInputEvent方法来实现的。

```
InputManager.getInstance().injectInputEvent(me,  
InputManager.INJECT_INPUT_EVENT_MODE_WAIT_FOR_RESULT)
```

所以2.3版本的Monkey在4.0版本的Android手机上运行不起来的。

4.5 本章小结

Monkey是Android测试中常用的一个稳定性测试工具，掌握Monkey工具本身的使用方法是简单的。但是真正能深入了解Monkey的代码实现逻辑，并且具备优化Monkey能力的人，却是少之又少。通过本章，读者不仅仅可以学习到Monkey的一些基本知识和基本使用方法，还可以通过对Monkey代码逻辑和扩展实例的学习，有所启发，掌握新的自动化测试的方案。

第5章 UIAutomator框架及实践

本章将从四个维度（图5-1）对UIAutomator自动化框架进行介绍，由浅入深地剖析其原理，讲述在TOS（Tencent OS，腾讯基于Android开发的手机系统）测试过程中的实践案例，围绕基础、原理、实战三方面，对于框架特性、适用场景进行分析，为二次开发提供思路及技巧。

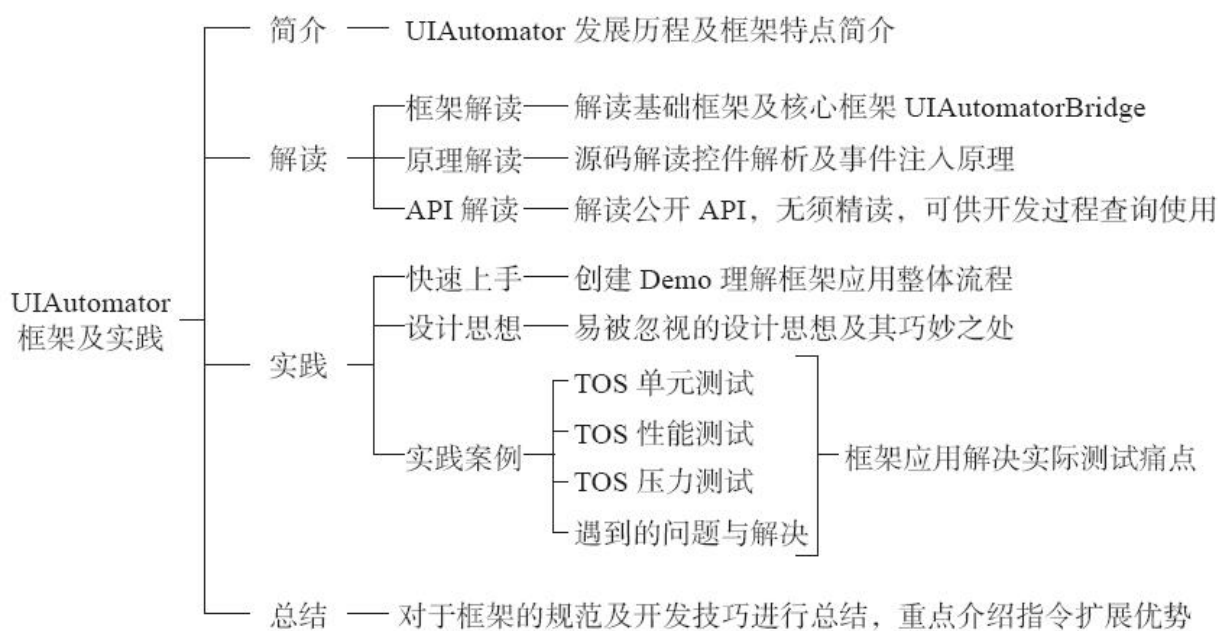


图5-1 本章知识结构图

UIAutomator作为自动化框架的一种，其初衷是更好地为测试服务，是确保产品质量的一种方法，而非目的。当测试的思想不再局限于框架本身时，我们可以更好地改造并利用其为测试服务，

UIAutomator具备的强大兼容性及应用的灵活性，可以很好地帮助我们解决自动化过程中的问题。

5.1 UIAutomator简介

测试领域根据代码是否可见可划分为黑盒测试、白盒测试，从执行方式的不同可划分为手工测试、自动化测试。在实际的移动端测试过程中，大部分采用敏捷开发的团队都在经历快速的需求迭代，为了适应其快速迭代的节奏，在测试选型上往往还是黑盒手工测试居多。而黑盒手工测试可以说是所有测试方法中最耗时、最无趣、最易出错的方法，考虑到白盒测试的成本太高，为了寻求更有效、更靠谱的测试方式，Android官方提供了一套黑盒UI自动化测试框架

UIAutomator。

Android官方关于UI测试有这么一段描述：在App的测试过程中，除了对Android的单独组件（如Activity、Service、Content、Provider）进行单元测试，测试应用运行时的界面行为也非常必要，通过测试界面行为来保证应用为用户一系列操作后，能够正确地呈现用户预期的结果。

UIAutomator是为数不多的Android官方支持的自动化框架之一，关于版本的选择可以参考更新特性。UIAutomator随着Android版本发布而更新，最早发布的版本为API Level 17。作为基于控件的自动化框架，UIAutoamtor的整体框架及API简单明晰，非常容易上手，发布后便受到了不少开发人员的好评，但仍有部分开发人员觉得不支持

resourceId检索控件有点儿可惜。官方在随后的Level 18中弥补了这一缺憾，至此，UIAutomator便在自动化测试领域占据了一席之地，满足了大部分开发人员的需求。

UIAutomator较于其他自动化框架有什么特性呢？笔者觉得可以用粗暴但灵活、简单可依赖来形容，其细数的优势有很多，概括起来有以下几点。

- 官方支持更新：** Android原生支持，测试依赖环境少，创建方便。
- 层次接口明晰：** 框架层次结果分明，API明晰，上手成本很低。
- 基于控件交互：** 支持Android原生控件解析，比坐标交互兼容性更强。
- 不依赖于源码：** 测试过程基于黑盒进行，对所有发行版本都可以测试。
- 事件等待优秀：** 在事件等待方面接口丰富，控制灵活精确，表现优秀。
- 支持跨进程测试：** 在自动化框架中，具备此特性的不多，测试范围在ROM层面。

较于其他框架来说，不足的地方就是UIAutomator仅支持API Level 17及以上，且控件解析仅支持Android原生控件，对于Web则无法解析。随着市场上的设备逐步更新，API Level 17以上的设备普及率也越来越高，这个缺点慢慢就不再是框架的短板了。而Web解析，确实是在框架选型的时候需要注意的地方，我们只能期盼着官方后续的更新，会考虑对这方面做出支持。

在技术选型方面，除了涉及Web的测试，UIAutomator基本上都可以帮用户实现。如果用户想做ROM层级的测试，或是App间协作需跨进程的测试，那UIAutomator将是非常好的选择；用户渴望写出简单易懂而功能强大的代码，也不妨选择一试，UIAutomator可以让用户在事件等待方面看到它优雅的一面；没有代码没关系，想要竞品对比也可以，单元测试、性能测试、压力测试，UIAutomator都可以成为用户的选择；简单而不简约，留给开发人员更自由的发挥空间，轻巧灵动，强大可靠，这就是UIAutomator。

5.2 UIAutomator解读

有个小故事，某程序员退休后决定练习书法，于是重金购买了文房四宝。一日，饭后突生雅兴，一番研墨拟纸，并点上上好檀香，定神片刻，泼墨挥毫，郑重地写下一行字：**Hello World!**

这句话对于程序员来讲再熟悉不过了，不是程序员都体会不到这句话的伟大所在。这几个简单的字符，往往意味着一个新的开始和一段新的征程，作为敲门砖一次次地打开新世界的大门。当程序员首次邂逅一项新技术的时候，大概都是迫不及待地想着这件事情吧。不过，在这里让我们先按捺住自己内心的小激动，为了后续能信手拈来玩得更好，避免一些先入为主的误区，还是先来解析一下UIAutomator的框架及工作原理。

5.2.1 UIAutomator框架解读

对于UIAutomator的框架，官网公开的分类只有下面九类，具体如下：

- UIAutomationSupport: UI测试信息拓展类。
- UIAutomatorTestCase: UI测试基类。
- UICollection: UI测试控件集合。
- UIDevice: UI测试设备抽象。
- UIObject: UI测试控件抽象。
- UIObjectNotFoundException: UI测试控件无法找到的异常。
- UIScrollable: UI测试可滚动控件。
- UISelector: UI测试控件选择器。
- UIWatcher: UI测试界面观察者。

对于普通的自动化测试来说，这九类已经基本能满足开发人员的所有测试需求了，甚至可以说只需要熟悉其中几个就可以开始做Android界面自动化测试了，但一些小众却比较实用的类也建议有所了

解。以下将从UIAutomator基础框架及稍深入一些的UIAutomatorBridge框架来介绍UIAutomator。

1.UIAutomator基础框架

UIAutomator基础框架作为开发人员入门知识，掌握后即可开始应用开发，如图5-2所示。

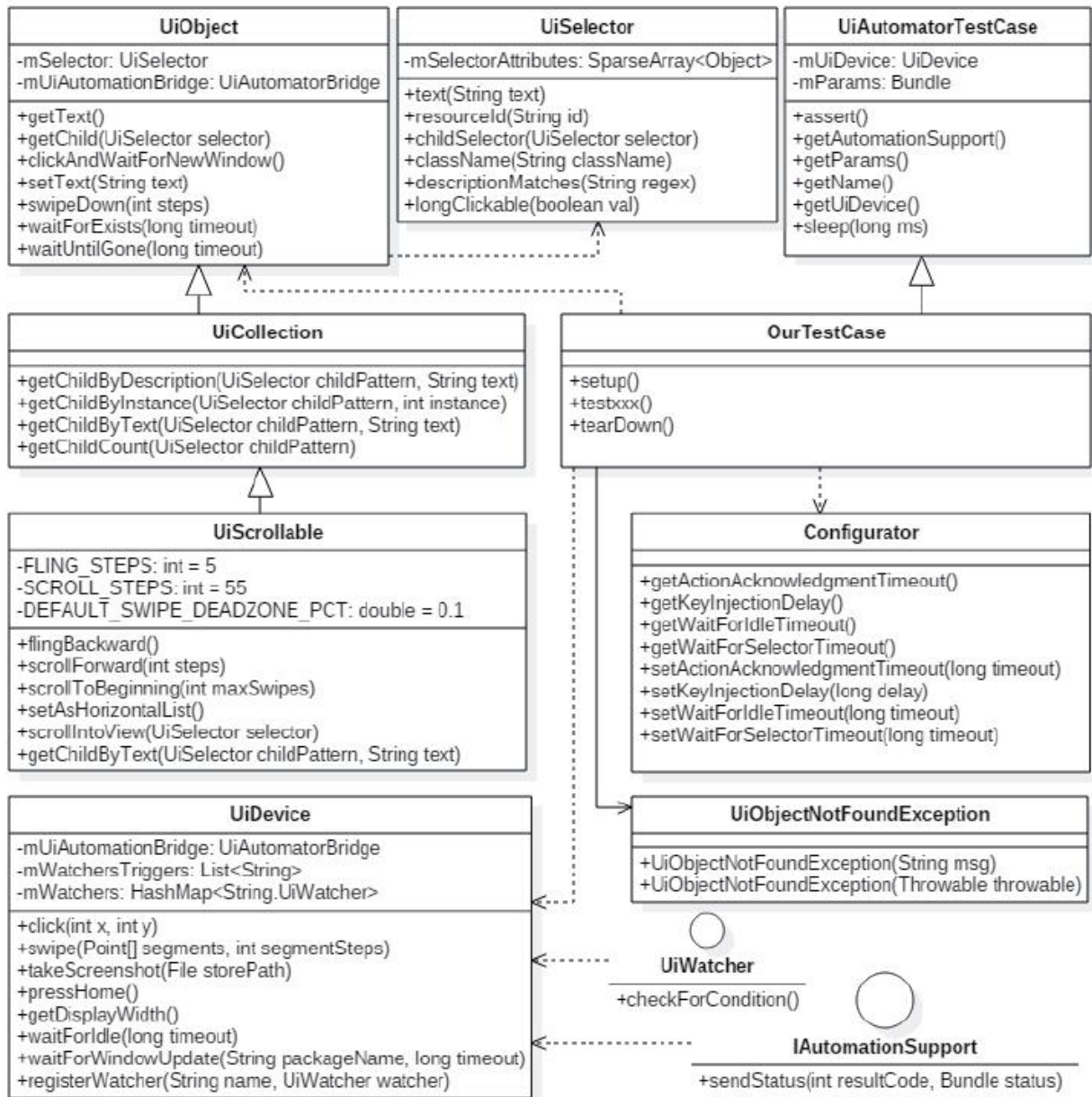


图5-2 UIAutomator基础框架

上图中除了OurTestCase为我们的自动化用例，框架，共10个类，可以分为一基类一配置、一设备一异常、两接口三控件外加一个选择器。其中使用最多的五大基础类为UiDevice、UiSelector、UiObject、

UiCollection、UiScrollable，接下来我们以功能分类，看一下这些类的具体作用。

1) 基类: UiAutomatorTestCase

作用：测试基类，所有测试用例都要继承于它，负责执行参数及用例信息获取。

描述：UIAutomator在执行的时候会进行有效性检查，只有继承于这个类的用例才可以被执行。这个基类继承于junit.framework.TestCase，执行setup、test、tearDown的Junit用例流程，遵循Assert（断言）的用例设计思想。

2) 配置: Configurator

作用：配置基础类，用以控制测试过程的事件等待超时、控件可见超时等。

描述：Configurator这个类容易被忽视，其保存了测试过程中关于事件注入的超时参数。通过设置这些参数可以控制操作的一些时长，比如通过修改控件可见超时保证控件匹配时获得更多时间，通过修改事件等待时长模拟快速双击，等等。

3) 设备: UiDevice

作用：设备封装类，测试过程获取设备信息、与设备交互。

描述：UiDevice是对当前测试设备的抽象，通过它可以获取设备的屏幕宽高、设备型号等信息，使用它可以向设备发送指令，比如截图、亮灭屏、点击、滑动等。

4) 异常：UiObjectNotFoundException

作用：异常处理机制，在预期控件不存在时向上抛出。

描述：当向预期控件发出操作指令，控件不存在时向上抛出，该框架唯一异常。

5) 接口：UiWatcher

作用：界面观察者，处理弹窗中断逻辑。

描述：UiWatcher其实是一个接口，实现该接口的实例可以向UiDevice注册作为界面观察者。在测试过程中，一旦有其他应用界面弹窗跳出，为了避免测试中断，该实例对应的接口方法就会被回调，用以处理中断弹窗以便测试继续进行。

6) 接口：IAutomationSupport

作用：测试辅助支撑类，用于发送测试状态。

描述：该类用的频率很低，可以向测试过程中的用例发送状态及传值。

7) 选择器：UiSelector

作用：控件选择器，利用控件属性描述目标控件，供控件匹配使用。

描述：UiSelector用于描述目标控件，只是作为属性信息的转储。在界面解析的时候，利用存储的属性对控件进行约束，过滤出我们想要的控件。在自动化测试过程中，UiObject拥有其作为成员变量，使用非常广泛，我们需要做的是，利用其属性描述来约束控件的唯一性。

8) 控件：UiObject

作用：所有控件抽象，用以表示一个Android控件。

描述：类似于Java中的Object，它是所有控件的超类，UIAutomator中关于控件的抽象程度很高，ListView、TextView、Button等，都用UiObject来表示，所以UiObject也包含了绝大部分控件的操作，包括点击、文字输入、拖拽等。无控件，不自动化，在自动化编写过程中，这个类使用的频率是最高的。

9) 控件：UiCollection

作用：控件集合，用于控件遍历。

描述：继承于UiObject，表示符合同一条件的控件集合，拓展了获取界面子节点元素的方法。当界面存在多个控件而无法用UiSelector描述目标控件的唯一性，或需要对界面元素进行遍历操作时，可以使用UiCollection来进行。

10) 控件：UiScrollable

作用：滚动控件，当目标控件存在于屏幕之外时使用。

描述：继承于UiCollection，使用该控件描述界面滑动列表，当目标控件存在于可见范围之外时，可以使用getChild系列方法来获取，UiScrollable会自动完成滑动操作以遍历列表里的所有元素。

综上所述，在实际自动化过程中，使用UiSelector对目标控件进行描述，根据需要选择三个控件类别得到目标控件实例，通过实例与控件进行交互，或者通过UiDevice对设备进行交互，期间可以通过Configurator修改配置，通过UiWatcher处理弹窗中断，捕获异常或是使用Assert来对用例结果进行判断。

2.UiAutomatorBridge框架

作为基于控件的自动化测试框架，有两件比较重要的事情，即界面解析和事件注入。界面解析以获取目标控件，事件注入以完成操作

交互。在UIAutomator框架中要了解这两件事情是怎样完成的，就不得不说到UiAutomatorBridge。

在UIAutomator自动化测试过程中，界面解析和事件注入最终都依赖于UiAutomation来完成，而UiAutomatorBridge相当于UiAutomation的代理，作为两者之间调用的桥梁，几乎所有事件的处理及交互，都经过UiAutomatorBridge进行派发。

UiAutomatorBridge框架如图5-3所示。

图5-3所示为与UiAutomatorBridge相关的核心类，其中事件起源为UiDevcie，UiAutomatorBridge作为代理，向UiAutomation派发事件取得数据，具体事务管理由QueryController、Interaction-Controller与ShellUiAutomatorBridge来完成。

·UiAutomatorBridge与UiDevice的关系：UiDevice成员变量中拥有UiAutomatorBridge的实例，视其为UiAutomation的代理，所有与界面解析及事件注入相关的事务，都调用UiAutomatorBridge来进行。

·UiAutomatorBridge与QueryController、InteractionController的关系：UiAutomatorBridge有两个助手，即QueryController与InteractionController，QueryController负责界面解析事务（把UiSelector转换成AccessibilityNodeInfo），而InteractionController负责事件注入事

务。当UiAutomatorBridge从UiDevice获取指令后，并不是直接与UiAutomation进行交互，而是根据事务属性，调用对应的类去执行。

·UiAutomatorBridge与ShellUiAutomatorBridge的关系：在UiAutomatorBridge中，getRotation、isScreenOn等方法是没有具体的实现的，为什么呢？在UiAutomatorBridge中，大部分方法都是需要调用UiAutomation才能完成的，当然也有部分不需要调用UiAutomation的方法。为了代码有更好的维护性，这部分方法就被抽取出来，在ShellUiAutomatorBridge中实现。

·UiAutomatorBridge与UiAutomation的关系：UiAutomatorBridge持有UiAutomation的实例作为成员变量，QueryController及InteractionController在执行事务的时候，会通过UiAutomatorBridge来获得UiAutomation的实例进行调用。

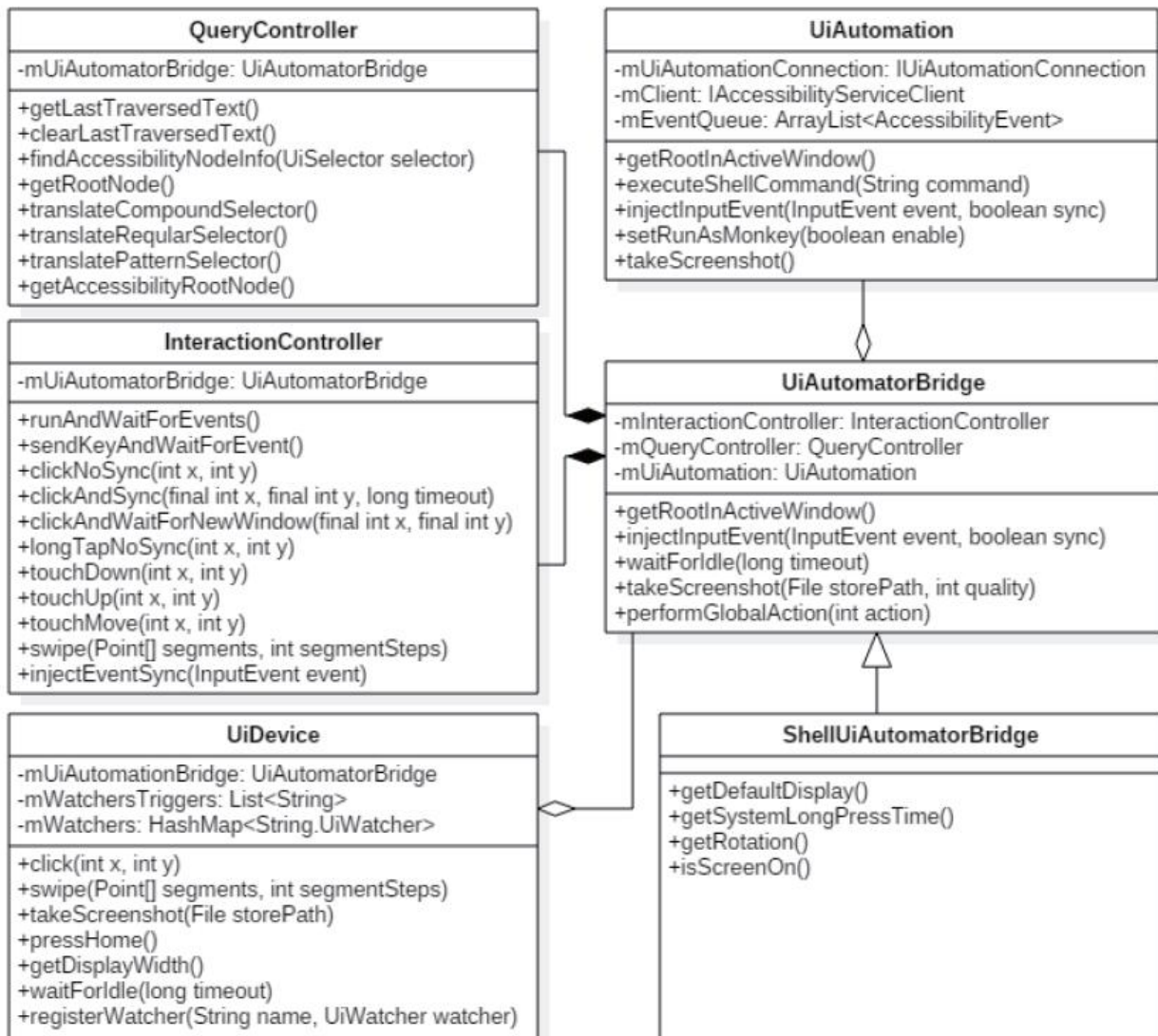


图5-3 UiAutomatorBridge框架



提示 在实际的自动化过程中，UiDevice中持有的实例对象并不是UiAutomatorBridge，而是其子类ShellUiAutomatorBridge。

5.2.2 UIAutomator原理解读

刚开始接触UIAutomator的时候，为了快速上手赶进度，只是学习框架怎么使用，对于一些存在的问题及现象，并没有进行深入的了解。比如，为什么在UIAutomator执行过程中整体感觉会比较慢？为什么在自动化执行过程中，logcat中会不断看到getText（）的日志输出？为什么明明调用了UiScrollable的scrollForward但没有作用.....

那时候觉得描述当前页面滑动列表的写法就一定是这样的：

```
UiScrollable mList = new UiScrollable(new UiSelector().scrollable(true));
```

最关键的是，一定要有scrollable（true），后面在做MIUI系统自动化对比测试的时候，突然发现界面解析出来的滑动列表控件属性中scrollable为false（MIUI系统屏蔽该属性），才明白通过其他方法也可以获得滑动控件进行使用。

来看一段简单的代码，如代码清单5-1所示，原意很简单，只是想寻找界面中text文案为“确定”的按钮，如果按钮存在，单击它。对于一个Java程序员来讲，这样的代码应该习以为常了，申请一个对象，如果对象不为空，则对其进行操作，再正常不过了。

代码清单5-1 单击确定按钮

```
/**
 * 单击确定按钮

 * @throws UiObjectNotFoundException
 *         UiObjectNotFoundException
 */
public void clickPositiveButton() throws UiObjectNotFoundException {
    UiObject mPositiveButton = new UiObject(new UiSelector()
        .className(Button.class).text("确定

"))
    );
    if(null != mPositiveButton)
        mPositiveButton.click();
}
```

那么问题来了，这里判断了`null != mPositiveButton`，再调用`mPositiveButton`的`click`方法，就能确保用例执行顺利，不抛出异常了吗？答案是否定的，如果当前页面不存在“确定”按钮，程序在执行`mPositiveButton.click()`的时候仍然会抛出`UiObjectNotFoundException`。有点儿不明白，为什么呢？这里`mPositiveButton`应该不为空才对。

通常来讲，这里很可能会有一个先入为主的想法，就是当`mPositiveButton`完成`new`的动作后就已经完成了当前界面的解析并找到了这个控件，将其赋予了`mPositiveButton`这个变量。这样的话，按上面代码的逻辑，确实不应该抛出`UiObjectNotFoundException`。好的，带着疑问，让我们来看一下实际的过程是怎么样的，我们找到`UiObject.java`中关于构造方法的代码，如代码清单5-2所示。

代码清单5-2 UiObject.java构造方法

```
71  /** UiObject.java
72   * Constructs a UiObject to represent a view that matches
73   * the specified selector criteria.
74   * @param selector
75   * @since API Level 16
76   */
77  public UiObject(UiSelector selector) {
78      mSelector = selector;
79  }
```

从上面代码中可以看出，在UiObject申请实例的时候，只是把参数selector赋予了成员变量mSelector，其他什么动作也没有发生。这样看来，上面“null != mPositiveButton”的判断基本是白写了，因为mPositiveButton基本上可以认为申请实例一定会成功。那么界面解析的过程在哪里呢？

剩下的唯一语句，就只有“mPositiveButton.click ()；”这一句了，估计这里面大有文章。在这里我们不妨以其为例，来解读一下UIAutomator界面解析和事件注入的过程，提取大致的流程，画出时序图，如图5-4所示。

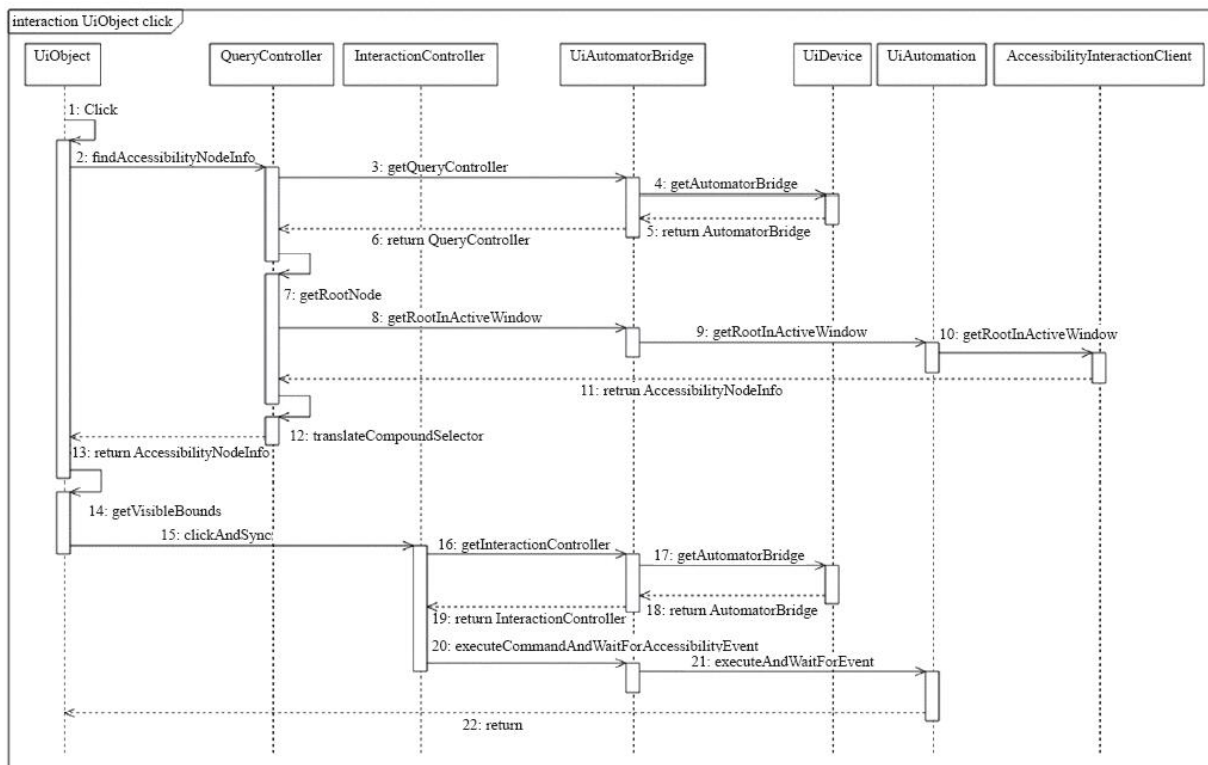


图5-4 UIAutomatorBridge框架

我们从代码层面来解析这个过程，当mPositiveButton调用click（）方法时，源码如代码清单5-3所示。

代码清单5-3 UiObject.click方法

```

380  /** UiObject.java
381   * Performs a click at the center of the visible bounds of
382   * the UI element represented by this UiObject.
383   *
384   * @return true id successful else false
385   * @throws UiObjectNotFoundException
386   * @since API Level 16
387   */
388  public boolean click() throws UiObjectNotFoundException {
389      Tracer.trace();
390      AccessibilityNodeInfo node = findAccessibilityNodeInfo(
391          mConfig.getWaitForSelectorTimeout());
392      if(node == null) {
393          throw new UiObjectNotFoundException(getSelector().toString());
394      }
395      Rect rect = getVisibleBounds(node);
  
```

```
395         return getInteractionController().clickAndSync(rect.centerX(),
396                 rect.centerY(), mConfig.getActionAcknowledgmentTimeout());
397     }
```

从上面的代码可以看到，390行先是调用`findAccessibilityNodeInfo`获得控件的节点信息，394行再根据节点信息获得控件边框信息，最后395行调用`clickAndSync`根据坐标进行点击，其中获得节点信息代码`findAccessibilityNodeInfo`如代码清单5-4所示。

代码清单5-4 `UiObject.findAccessibilityNodeInfo`方法

```
155  /** UiObject.java
156   * Finds a matching UI element in the accessibility hierarchy, by
157   * using the selector for this UiObject.
158   * @param timeout in milliseconds
159   * @return AccessibilityNodeInfo if found else null
160   */
161  protected AccessibilityNodeInfo findAccessibilityNodeInfo(
162      long timeout) {
163      AccessibilityNodeInfo node = null;
164      long startMills = SystemClock.uptimeMillis();
165      long currentMills = 0;
166      while (currentMills <= timeout) {
167          node = getQueryController().
168              findAccessibilityNodeInfo(getSelector());
169          if (node != null) {
170              break;
171          } else {
172              // does nothing if we're reentering another runWatchers()
173              UiDevice.getInstance().runWatchers();
174          }
175          currentMills = SystemClock.uptimeMillis() - startMills;
176          if (timeout > 0) {
177              SystemClock.sleep(WAIT_FOR_SELECTOR_POLL);
178          }
179      }
180      return node;
181  }
```

在`findAccessibilityNodeInfo`方法中，关键语句为168行，调用`getQueryController`取得`QueryController`（`UiDevice` → `getAutomatorBridge` → `getQueryController`），再调用

QueryController.findAccessibilityNodeInfo方法获得控件节点信息，如代码清单5-5所示。

代码清单5-5 QueryController.findAccessibilityNodeInfo方法

```
141     protected AccessibilityNodeInfo findAccessibilityNodeInfo(UiSelector
142         selector, boolean isCounting) {
143         mUiAutomatorBridge.waitForIdle();
144         initializeNewSearch();
145
146         if (DEBUG)
147             Log.d(LOG_TAG, "Searching: " + selector);
148
149         synchronized (mLock) {
150             AccessibilityNodeInfo rootNode = getRootNode();
151             if (rootNode == null) {
152                 Log.e(LOG_TAG, "Cannot proceed when root node is null.");
153                 return null;
154             }
155
156             // Copy so that we don't modify the original's sub selectors
157             UiSelector uiSelector = new UiSelector(selector);
158             return translateCompoundSelector(uiSelector, rootNode,
159                 isCounting);
160         }
```

在QueryController的findAccessibilityNodeInfo方法中，主要有两个步骤，先调用150行getRootNode得到当前的根节点，然后158行再使用selector根据根节点遍历得到目标控件节点信息。获取根节点的过程如代码清单5-6、代码清单5-7所示。

代码清单5-6 QueryController.getRootNode方法

```
165     * @return null if no root node is obtained
166     */
167     protected AccessibilityNodeInfo getRootNode() {
168         final int maxRetry = 4;
169         final long waitInterval = 250;
170         AccessibilityNodeInfo rootNode = null;
171         for(int x = 0; x < maxRetry; x++) {
```

```
172         rootNode = mUiAutomatorBridge.getRootInActiveWindow();
173         if (rootNode != null) {
174             return rootNode;
175         }
176         if(x < maxRetry - 1) {
177             Log.e(LOG_TAG, "Got null root node - Retrying...");
178             SystemClock.sleep(waitInterval);
179         }
180     }
181     return rootNode;
182 }
```

代码清单5-7 UiAutomatorBridge.getRootInActiveWindow方法

```
65     public AccessibilityNodeInfo getRootInActiveWindow() {
66         return mUiAutomation.getRootInActiveWindow();
67     }
```

在getRootNode方法中，可以看到172行调用的是UiAutomatorBridge的getRoot-InActiveWindow（）方法来获取得到Root节点，而最终UiAutomatorBridge调用mUiAutomation.getRootInActiveWindow（）方法来获取。顺利取得根节点的信息后，QueryController再调用translateCompoundSelector方法把selector转换成对应的AccessibilityNodeInfo。

至此，界面解析的过程结束。接下来是事件注入的过程，如代码清单5-3中394、395行，通过返回的节点信息取得边框信息，调用getInteractionController（）获得InteractionController实例（UiDevice → getAutomatorBridge → getInteractionController），通过clickAndSync来完成点击事件注入，而clickAndSync方法是通过调用runAndWaitForEvents来实现的，如代码清单5-8所示。

代码清单5-8 InteractionController.runAndWaitForEvents方法

```
157 private AccessibilityEvent runAndWaitForEvents(Runnable command,
158         AccessibilityEventFilter filter, long timeout) {
159
160     try {
161         return mUiAutomatorBridge;
162         executeCommandAndWaitForAccessibilityEvent(
163             command, filter, timeout);
164     } catch (TimeoutException e) {
165         Log.w(LOG_TAG, "runAndWaitForEvent timedout waiting events");
166         return null;
167     } catch (Exception e) {
168         Log.e(LOG_TAG, "exception executeWaitForAccessibilityEvent", e);
169         return null;
170     }
```

实际上，InteractionController在161和162行处调用了UiAutomatorBridge去执行事件注入executeCommandAndWaitForAccessibilityEvent。在该方法中，事件被封装成Runnable-Command，由于该调用来自clickAndSync，跟进可以发现这里使用的是clickRunnable，如代码清单5-9所示。

代码清单5-9

UiAutomatorBridge.executeCommandAndWaitForAccessibilityEvent方法

```
102 public AccessibilityEvent executeCommandAndWaitForAccessibilityEvent
103     (Runnable command, AccessibilityEventFilter filter, long timeoutMillis)
104     throws TimeoutException {
105     return mUiAutomation.executeAndWaitForEvent(command,
106         filter, timeoutMillis);
106 }
```

UiAutomatorBridge最终指向UiAutomation，所有事件最终都交给它去执行。到这里我们大体理清了UIAutomator界面解析和事件注入的流

程，可以稍做总结如下：

- UIAutomator界面解析、事件注入均由UiAutomation来完成。
- UiObject对象可以理解成使用时才被实例化，每次调用都会重新解析实例化。
- UIAutomator的操作事件非基于控件，最终都转换为坐标执行。
- 控件遍历过程中每次只返回一个节点信息而不是控件树，效率比较低。

5.2.3 UIAutomator API解读

分析完UIAutomator的框架和原理后，接下来我们通过分类来解读其API，了解常用的API及其使用场景，首先从整体来看一下UIAutomator执行时的指令：

```
# adb shell uiautomator runtest <jar> -c <test_class_or_method> [options]
```

UIAutomator为Android自带的可执行程序，用来执行与UIAutomator自动化测试相关事宜，一般通过adb shell来调用，当然，在手机端使用runTime执行也是可以的。下面详细介绍指令中的几个参数，了解它们分别是什么意思及是怎么使用的。

(1) runtest。runtest是执行测试的关键指令，对应的指令还有一个“events”（用于打印accessibility事件到控制台，直至设备断开连接）。由于events使用的频率较低，以下的参数介绍均基于runtest指令来进行。

(2) <jar>。<jar>是紧跟在runtest后面的参数，用于指定需要执行的测试用例所在的jar包名称，使用相对路径（UIAutomator的jar包放置于/data/local/tmp/目录下），可以多指定。

若只存在一个测试jar包A.jar，直接写代码如下：

```
# adb shell uiautomator runtest A.jar -c <test_class_or_method> [options]
```

若A.jar测试过程中引入了第三方jar包B.jar，则并列写，中间使用空格符分隔即可，代码如下：

```
# adb shell uiautomator runtest A.jar B.jar -c <test_class_or_method> [options]
```

(3) -c<test_class_or_method>。其用以指定测试具体的Case，参数指定时需要写全量路径（包名.class#method），可以有三种指定方式。此处假设A.jar中存在两个测试类C.java、D.java，分别存在测试方法C.testC_e、C.testC_f、D.testD_g、D.testD_h，则以下不同的指定方式会有不同的效果。

什么也不指定：会执行指定jar包中所有的测试用例（以test开头的方法，通过反射进行调用，无执行顺序），在此会执行testC_e、testC_f、testD_g、testD_h。

```
# adb shell uiautomator runtest A.jar
```

指定测试用例类名（-c class）：会执行指定类下的所有用例，如果有多个类可以并列指定，中间用空格符分隔开，如以下指令会运行testC_e、testC_f。

```
# adb shell uiautomator runtest A.jar -  
c com.tencent.uiAutotest.C
```

指定测试用例类名加方法名（-c class#method）：会执行指定类下指定的测试方法，同样多个方法之前可以使用空格分开并列指定，如下指令会运行testC_e、testD_g。

```
# adb shell uiautomator runtest A.jar -  
  
c com.tencent.uiatotest.C#testC_e  
-  
  
c com.tencent.uiatotest.D#testD_g
```

（4）[options]。[options]用以指定测试过程的特殊参数，比如后台挂起测试、向测试过程传递测试参数、调试当前测试用例或者dump当前xml视图以保存现场进行后续分析，具体的用法见表5-1。

表5-1 UIAutomator执行options参数解析

选项名称	功能描述
--nohup	指定后台挂起运行，若没有该参数，通过 adb shell 执行的 UIAutomator 在设备断开连接的时候就会停止运行，使用该参数的效果等价于使用 “&” 挂起
-e <key> <value>	给执行过程指定参数，以键值对的方式动态封装成 Bundle 供测试过程使用，在 UiAutomatorTestCase 中使用 getParams() 可获得，可以多项指定
-e debug [true false]	本质上和上面的 -e 是一样的用法，其特别的地方在于 debug 为 UIAutomator 指令中的保留字，指定该参数可以打开 UIAutomator 执行过程的调试端口，默认调试模式为关闭
dump [file]	dump 当前界面的 xml 至文件中，通常用于保留当前页面供调试分析，file 为保存的文件路径，不指定时默认为 /storage/sdcard0/window_dump.xml

1.UiAutomatorTestCase

作为所有测试用例的超类，UiAutomatorTestCase继承于junit.framework.TestCase，遵循setUp、test、tearDown的测试流程，支

持断言使用，负责基础的框架支持，包含执行过程的参数获取、实例获取及断言使用。对应API解析如下：

（1）参数获取。UiAutomatorTestCase参数获取API见表5-2。

表5-2 UiAutomatorTestCase参数获取API

返回值	方法及说明
Bundle	<code>getParams()</code> 在执行 UIAutomator 时，可以使用 <code>-e <key> <value></code> 给执行过程指定参数，所有键值对存储于参数 Bundle 中，在测试过程中通过 <code>getParams()</code> 得到该 Bundle

（2）实例获取。UiAutomatorTestCase实例获取API见表5-3。

表5-3 UiAutomatorTestCase实例获取API

返回值	方法及说明
UiDevice	<code>getUiDevice()</code> 取得当前设备实例，等效于使用 <code>UiDevice.getInstance()</code>
IAutomationSupport	<code>getAutomationSupport()</code> 取得 UIAutomator 实现的 IAutomationSupport 实例用以向结果添加 INSTRUMENTATION_ STATUS 标识的日志信息

（3）流程执行。UiAutomatorTestCase流程执行API见表5-4。

表5-4 UiAutomatorTestCase流程执行API

返回值	方法及说明
void	<code>setup()</code> 测试前环境准备，在同一个类中于每个 <code>testCase</code> 前执行，一般被用户覆写，将测试时的初始化准备放置于该方法内进行

(续)

返回值	方法及说明
void	<code>sleep(long ms)</code> 休眠指定时间，等效于使用 <code>SystemClock.sleep(long ms)</code>
void	<code>tearDown()</code> 测试后收尾工作，在同一个类中于每个 <code>testCase</code> 后执行，一般被用户覆写，用以处理执行结果、保存数据、还原测试前环境

(4) 断言支持。UiAutomatorTestCase断言支持API见表5-5。

表5-5 UiAutomatorTestCase断言支持API

返回值	方法及说明
void	<code>assertXXX()</code> 系列 继承于 <code>junit.framework.Assert</code> ，常用断言来检查当前的测试环境状态，如果断言失败，则认为 <code>case</code> 执行失败， <code>case</code> 执行中断，进入 <code>tearDown()</code>

2.UiDevice

UiDevice用来与测试设备进行交互，获取设备信息、发送操作指令及保存截图布局等状态，根据其API功能的不同，以下分几个方面简单介绍其常用的功能。

(1) 事件操作相关。向设备发送按钮点击事件，封装了部分常用的按钮，但所有按钮事件都可以通过`pressKeyCode (int keyCode)`这个方法等效指定，见表5-6。

表5-6 UiDevice事件操作API

返回值	方法及说明
boolean	<code>pressBack()</code> / <code>pressHome()</code> / <code>pressMenu()</code> / <code>pressSearch()</code> 单击返回键 / Home 键 / 菜单键 / 搜索键
boolean	<code>pressKeyCode(int keyCode)</code> 向设备发送事件 <code>keyCode</code> ，具体事件可参见 <code>KeyEvent</code>

(2) 屏幕操作相关。向设备发送屏幕操作事件，包含点击、拖曳、修改设备屏幕状态（亮灭屏、屏幕方向），其中双击可以使用 `click` 进行组合，多个点连续滑动可以使用 `swipe` (`Point[]segments`, `int segmentSteps`)，见表5-7。

表5-7 UiDevice屏幕操作API

返回值	方法及说明
boolean	<code>click(int x, int y)</code> 单击屏幕坐标点 (x, y)，坐标原点从屏幕左上角开始
boolean	<code>drag(int startX, int startY, int endX, int endY, int steps)</code> 从 (startX, startY) 向 (endX, endY) 拖曳，步长为 steps
(续)	
返回值	方法及说明
boolean	<code>swipe(int startX, int startY, int endX, int endY, int steps)</code> 从 (startX, startY) 向 (endX, endY) 滑动，步长为 steps
boolean	<code>swipe(Point[] segments, int segmentSteps)</code> 按住不动顺序完成点间的滑动，步长为 segmentSteps 多个点之前连续滑动可以使用该 API 来实现
void	<code>sleep()</code> 设备灭屏，进入休眠状态
void	<code>wakeUp()</code> 唤醒设备，一般配合 <code>isScreenOn</code> 查询状态进行
void	<code>setOrientationLeft()</code> / <code>setOrientationNatural()</code> / <code>setOrientationRight()</code> 设置屏幕向左旋转 90 度 / 恢复自然角度 / 向右旋转 90 度

(3) 快捷开关相关。封装Android通用快捷操作，包含打开通知栏、快速设置、最近任务栏，其并非通过模拟界面点击，而是通过服

务事件调用，所以对不同的ROM都有较好的兼容性，推荐使用，见表5-8。

表5-8 UiDevice快捷开关API

返回值	方法及说明
boolean	pressRecentApps () 打开最近任务界面（多任务切换页面）
boolean	openNotification () 打开通知栏
boolean	openQuickSettings () 打开快捷设置栏

（4）设备截图&监听相关。截图可指定缩放比例用户截图质量，质量越高，需要的时间越长。通常用于保留问题现场，而注册监听则是为测试过程置入界面观察者，以处理中断弹窗确保测试的顺利进行，见表5-9。

表5-9 UiDevice设备截图&监听API

返回值	方法及说明
boolean	takeScreenshot(File storePath) 截取当前屏幕截图，保存至指定文件
boolean	takeScreenshot(File storePath, float scale, int quality) 截取屏幕截图，scale 为缩放比例，quality 为截图质量
boolean	registerWatcher(String name, UiWatcher watcher) 注册界面观察者，以处理中断弹窗，name 作为移除标识
boolean	removeWatcher(String name) 移除界面观察者，name 与注册时对应

（5）属性获取。UiDevice属性获取API见表5-10。

表5-10 UiDevice属性获取API

返回值	方法及说明
int	<code>getDisplayHeight()</code> 获取当前屏幕高度
int	<code>getDisplayWidth()</code> 获取当前屏幕宽度
String	<code>getCurrentActivityName()</code> 获取当前 Activity 名
String	<code>getCurrentPackageName()</code> 获取当前页面所属应用包名
boolean	<code>isScreenOn()</code> 当前屏幕是否是亮起状态

(6) 视图相关。UiDevice视图相关API见表5-11。

表5-11 UiDevice视图相关API

返回值	方法及说明
void	<code>dumpWindowHierarchy (File dest)</code> dump 当前窗口视图的布局到指定文件
void	<code>dumpWindowHierarchy(String fileName)</code> dump 当前窗口视图的布局到指定文件
void	<code>dumpWindowHierarchy(OutputStream out)</code> dump 当前窗口视图的布局到指定文件
String	<code>getLastTraversedText()</code> 获取上一次设置的 text 内容
void	<code>clearLastTraversedText()</code> 清除上一次设置的 text 内容

(7) 事件等待。UiDevice事件等待API见表5-12。

表5-12 UiDevice事件等待API

返回值	方法及说明
void	<code>waitForIdle()</code> / <code>waitForIdle(long timeout)</code> 无限时等待当前应用空闲 / 等待应用空闲，若超时则不再等待
boolean	<code>waitForWindowUpdate(String packageName, long timeout)</code> 等待指定包名的任意窗口更新，若超时则不再等待

3.UiSelector

UiSelector用来描述目标控件的特征，所有方法调用后返回的都是UiSelector，所以支持链式调用填充多个属性，按匹配的策略大体可以分为以下几种类型。

(1) 完全匹配。UiSelector完全匹配API见表5-13。

表5-13 UiSelector完全匹配API

返回值	方法及说明
UiSelector	checked(boolean val) / selected(boolean val) 目标控件是否可以被勾选，一般为 checkBox / 是否被选中
UiSelector	enabled(boolean val) / clickable(boolean val) / longClickable(boolean val) 目标控件是否可用 / 响应点击 / 响应长按
UiSelector	className(Class<T> type) / className(String className) 指定目标控件的类型为 type
UiSelector	description(String desc) 指定目标控件的描述为 desc
UiSelector	focusable(boolean val) / focused(boolean val) 目标控件是否可被聚焦 / 是否正被聚焦
UiSelector	index(int index) 指定目标控件的下标为 index
UiSelector	instance(int instance) 指定目标控件为符合条件的第 N 个实例，通常在集合遍历时使用
UiSelector	packageName(String name) 指定目标控件的包名为 name
UiSelector	text(String text) 指定目标控件的文案为 text
UiSelector	scrollable(boolean val) 目标控件是否可以滚动，当 listView 为一页时实际上为 false
UiSelector	resourceId(String id) 指定目标控件的 ID 为 id

(2) 部分包含。UiSelector部分包含API见表5-14。

表5-14 UiSelector部分包含API

返回值	方法及说明
UiSelector	descriptionStartsWith(String desc) 指定目标控件描述以 desc 开头
UiSelector	descriptionContains(String desc) 指定目标控件描述包含 desc
UiSelector	textStartsWith(String text) 指定目标控件文案以 text 开头
UiSelector	textContains(String text) 指定目标控件文案包含 text

(3) 正则匹配。UiSelector正则匹配API见表5-15。

表5-15 UiSelector正则匹配API

返回值	方法及说明
UiSelector	textMatches(String regex) / descriptionMatches(String regex) 指定目标控件文案 / 描述匹配 regex
UiSelector	packageNameMatches(String regex) 指定目标控件包名匹配 regex
UiSelector	classNameMatches(String regex) 指定目标控件类名匹配 regex
UiSelector	resourceIdMatches(String regex) 指定目标控件 ID 匹配 regex

(4) 父子关系。UiSelector父子关系API见表5-16。

表5-16 UiSelector父子关系API

返回值	方法及说明
UiSelector	childSelector(UiSelector selector) 指定目标控件拥有孩子节点匹配 selector
UiSelector	fromParent(UiSelector selector) 指定目标控件拥有父节点匹配 selector

4.UiObject

UiObject抽象的程度比较高，所有Android基础控件都可以用UiObject来表示，在自动化过程中用以完成信息获取及控件交互。

(1) 属性获取。UiObject属性获取API见表5-17。

表5-17 UiObject属性获取API

返回值	方法及说明
boolean	exists() 控件是否存在，控件不会 null 时不代表控件存在，需要调用此方法才可以确认控件是否在当前页面，交互前建议先调用此方法确认
Rect	getBounds() 获得控件的完整边框信息（含未可见部分）
Rect	getVisibleBounds() 获得控件的可见边框信息（不可见部分无效）
String	getContentDescription() 获得控件的描述信息
String	getPackageName() 获得控件的包名
boolean	isCheckedable() / isChecked() 控件是否可勾选 / 是否已经被勾选
(续)	
返回值	方法及说明
boolean	isFocusable() / isFocused() 控件是否可聚焦 / 是否为当前聚焦
boolean	isEnabled() / isClickable() / isLongClickable() / isScrollable() 控件是否可用 / 响应点击 / 响应长按 / 可滚动
boolean	isSelected() 控件是否已经被选中
String	getText() 获取控件的文案

(2) 控件获取。UiObject属性获取API见表5-18。

表5-18 UiObject属性获取API

返回值	方法及说明
int	<code>getChildCount()</code> 获取孩子控件的个数
UiObject	<code>getChild(UiSelector selector)</code> 在孩子节点中根据 <code>selector</code> 获取对应控件
UiObject	<code>getFromParent(UiSelector selector)</code> 在父节点（仅一级）下根据 <code>selector</code> 获取对应控件

（3）操作相关。UiObject相关操作API见表5-19。

表5-19 UiObject相关操作API

返回值	方法及说明
boolean	<code>click()</code> / <code>clickBottomRight()</code> / <code>clickTopLeft()</code> 点击控件中间 / 右下角 / 左上角
boolean	<code>clickAndWaitForNewWindow(long timeout)</code> 点击控件并且等待至有新窗口出现，如果超时间内有新窗口出现则继续往下走，如果没有，则一直等到超时后继续往下走，强烈建议使用
boolean	<code>longClick()</code> / <code>longClickBottomRight()</code> / <code>longClickTopLeft()</code> 长按控件中间 / 右下角 / 左上角
boolean	<code>swipeDown(int steps)</code> / <code>swipeUp(int steps)</code> 从控件中间向下滑动 / 向上滑动
boolean	<code>setText(String text)</code> / <code>clearTextFiled()</code> 设置控件文案 / 清除控件文案（EditText）

（4）事件等待。UiObject事件等待API见表5-20。

5.UiCollection

UiCollection继承于UiObject，用于表示符合同一UiSelector的控件集合，通常用于集合的遍历使用，较于UiObject多了四个方法，用于获取集合元素个数及指定元素。其集合元素数量可使用getChildCount获取，见表5-21。

表5-20 UiObject事件等待API

返回值	方法及说明
boolean	waitForExists(long timeout) 超时时间内等待控件出现，超时后则不再等待
boolean	waitUntilGone(long timeout) 超时时间内等待控件消失，超时后则不再等待

表5-21 UiCollection相关操作API

返回值	方法及说明
UiObject	getChildByDescription(UiSelector childPattern, String text) 根据 childPattern 及描述匹配返回子控件
UiObject	getChildByInstance(UiSelector childPattern, int instance) 根据 childPattern 匹配，取第 instance 个实例对象，同样也可以这么写 mUiCollection. getChild(new UiSelector.instance(i))
UiObject	getChildByText(UiSelector childPattern, String text) 根据 childPattern 及文案匹配返回子控件
int	getChildCount(UiSelector childPattern) 获得匹配 childPattern 的控件个数

6.UiScrollable

UiScrollable继承于UiCollection，用来表示可滚动控件。通常用于屏幕之外的控件检索，封装了滚动操作及自动滚动查找控件的操作，在实际的自动化过程中还是比较实用的。

(1) 状态相关。UiScrollable状态API见表5-22。

表5-22 UiScrollable状态API

返回值	方法及说明
int	getMaxSearchSwipes() / setMaxSearchSwipes(int swipes) 获取 / 设置检索最大滚动次数
void	setAsHorizontalList() / setAsVerticalList() 设置为水平列表 / 垂直列表
void	setSwipeDeadZonePercentage(double percentage) 设置击盲区的百分比，默认值 0.1

(2) 操作相关。UiScrollable相关操作API见表5-23。

(3) 控件获取。UiScrollable相关操作API见表5-24。

表5-23 UiScrollable相关操作API

返回值	方法及说明
boolean	flingBackward() / flingForward() 飞速向前 / 向后滑动 (step = 5)，较于 scroll 方法比较快
boolean	flingToBeginning(int maxSwipes) / flingToEnd(int maxSwipes) 快速飞滑至开始 / 结束，最大滑动次数为 maxSwipes
boolean	scrollBackward() / scrollForward() 以默认速度向前 / 向后滑动 (step = 55)
boolean	scrollToBeginning(int maxSwipes) / scrollToEnd(int maxSwipes) 以默认速度滚动至开始 / 结束，最大滑动次数为 maxSwipes
boolean	scrollDescriptionIntoView(String text) 滚动至描述为 text 的控件，如果没有，则遍历停留在列表尾部
boolean	scrollIntoView(Selector selector) 滚动至匹配 selector 的控件，如果没有，则遍历停留在列表尾部
boolean	scrollTextIntoView(String text) 滚动至文案为 text 的控件，如果没有，则遍历停留在列表尾部

表5-24 UiScrollable相关操作API

返回值	方法及说明
UiObject	<code>getChildByDescription(UiSelector childPattern, String text, boolean allowScrollSearch)</code> 查找描述为 <code>text</code> 且匹配 <code>childPattern</code> 的控件， <code>allowScrollSearch</code> 表示是否自动滚动查询，默认为 <code>true</code> ，为 <code>false</code> 时只查找当前页面
UiObject	<code>getChildByInstance(UiSelector childPattern, int instance)</code> 查找实例编号为 <code>instance</code> 且匹配 <code>childPattern</code> 的控件
UiObject	<code>getChildByText(UiSelector childPattern, String text, boolean allowScrollSearch)</code> 查找文案为 <code>text</code> 且匹配 <code>childPattern</code> 的控件， <code>allowScrollSearch</code> 表示是否自动滚动查询，默认为 <code>true</code> ，为 <code>false</code> 时只查找当前页面

7.Configurator

Configurator是UIAutomator的配置类，主要用于获取设置测试过程中的超时等待参数，通过修改参数可以调整操作速率，更加贴近想要模拟的用户操作场景。配置设置API见表5-25。

表5-25 配置设置API

返回值	方法及说明
long	<code>getActionAcknowledgmentTimeout()</code> 获取当前通用动作的超时时间
long	<code>getKeyInjectionDelay()</code> 获取当前键盘输入的超时时间
long	<code>getScrollAcknowledgmentTimeout()</code> 获取当前滚动动作的超时时间

(续)

返回值	方法及说明
long	<code>getWaitForIdleTimeout()</code> 获取当前等待应用空闲超时时间
long	<code>getWaitForSelectorTimeout()</code> 获取当前控件匹配的超时时间
Configurator	<code>setActionAcknowledgmentTimeout(long timeout)</code> 设定通用动作的超时时间
Configurator	<code>setKeyInjectionDelay(long delay)</code> 设定键盘输入的超时时间
Configurator	<code>setScrollAcknowledgmentTimeout(long timeout)</code> 设定滚动动作的超时时间
Configurator	<code>setWaitForIdleTimeout(long timeout)</code> 设定等待应用空闲超时时间
Configurator	<code>setWaitForSelectorTimeout(long timeout)</code> 设定控件匹配的超时时间

8.UiWatcher

UiWatcher是interface作为界面观察者处理测试过程所有的弹窗中断，接口仅存在一个方法即checkForCondition，实现后调用UiDevice进行注册，其生命周期是从注册的那一刻开始，直至测试结束或者调用UiDevice进行移除。如果用户想防止突然弹出的Crash弹框、闹钟、电话等阻塞测试的执行，UiWatcher还是非常实用的选择。对应API说明见表5-26。

表5-26 UiWatcher对应API说明

返回值	方法及说明
boolean	<code>checkForCondition()</code> 当进行控件匹配而当前又没有任何匹配控件时，framework 会遍历所有注册的 UiWatcher，回调此方法，以处理不可预料的弹框或者页面导致测试中断。如果回调中找到了匹配的情况并予以处理，则返回 true；如果未有任何匹配（情况在预料之外），则返回 false

5.3 UIAutomator实战

了解完UIAutomator的原理及常用API后，就可以动手开始我们的自动化进程了。接下来我们通过UIAutomator在实际测试运用过程中的一些案例及遇到的问题解决，来详细解说这个框架的一些设计思想及运用技巧（笔者建议提前安装ADT Bundle开发环境，以下开发均基于此环境进行，可以移步<https://developer.android.com/intl/zh-cn/sdk/index.html> 进行下载及安装学习）。

5.3.1 UIAutomator快速上手

UIAutomator测试项目的整体流程大体上可以分为以下几个步骤。

- (1) 分析应用UI界面元素，获取元素属性。
- (2) 创建测试用例，编码模拟用户操作过程。
- (3) 编译测试代码为测试jar包，push至终端。
- (4) 在设备上运行测试，查看测试结果。
- (5) 修改发现的BUG，修复并重新测试。

1.界面元素获取

UIAutomator提供了一个界面解析器供开发者使用，位于Android SDK/tools/目录下，建议配置至系统环境变量Path中，在后续开发过程经常用到。打开UIAutomator Viewer后，可以使用两种方法进行界面解析：一是使用UIAutomator runtest dump保存下来的xml文件；二是连接设备后直接点击左上角Device screenshot获取当前界面的解析结果。其中方法二较为常用。

如图5-5所示，左边为界面导航器，可以使用鼠标单击的方式选中目标控件。右上方为控件树，在此可以看到控件之间的层级关系及当前控件在控件树中的位置。右下方为当前控件的详细信息，在此可以获取目标控件的属性信息以供编码使用。

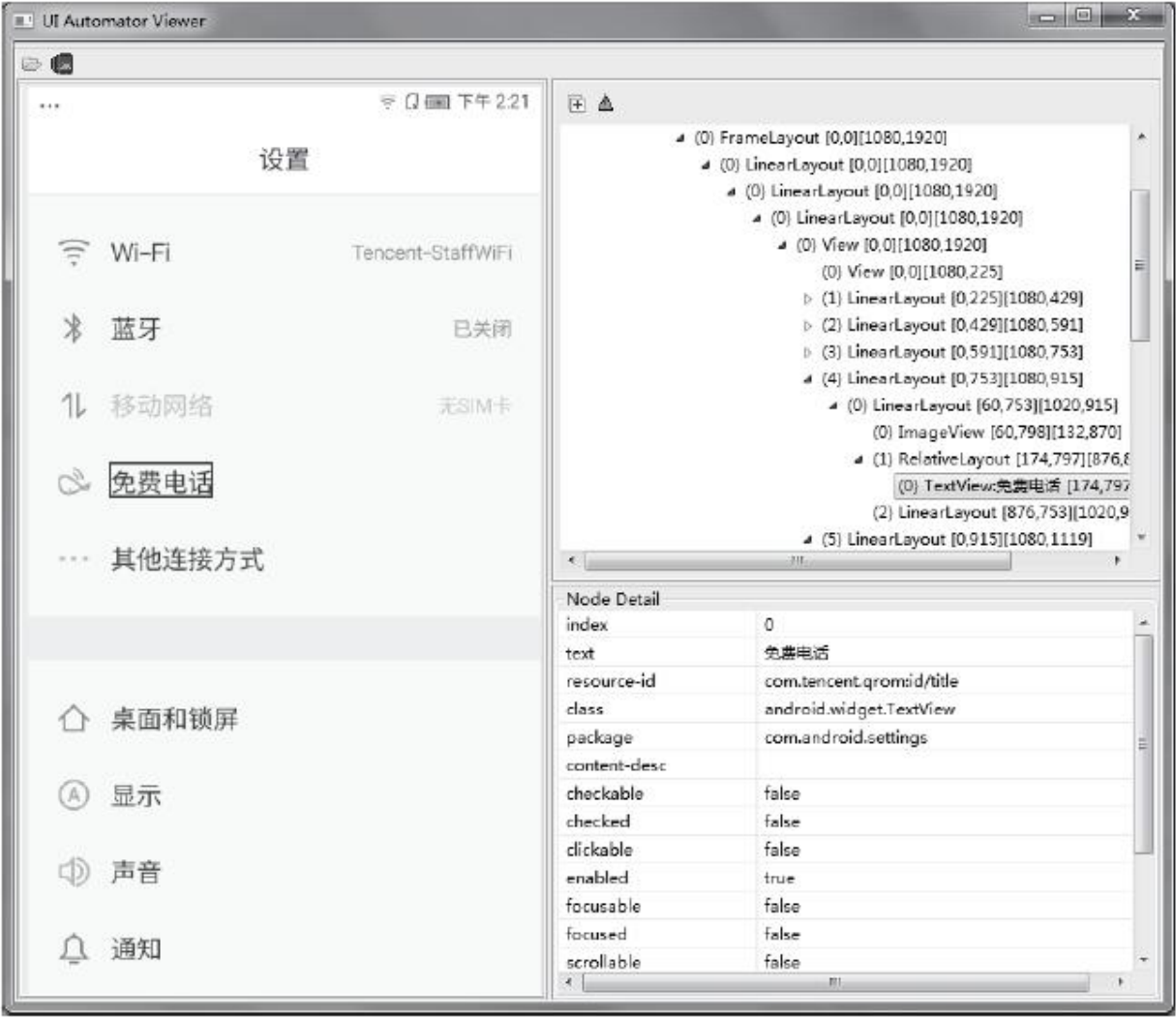


图5-5 UIAutomatorViewer界面

2.项目环境配置

UIAutomator自动化项目从创建Java项目开始（非Android项目），其所需要依赖的库有android.jar及uiautomator.jar，均位于/Android SDK/platforms/android-xx/目录下，其中xx表示目标API Level，建议使用19以上（新增UiSelector.resourceId系列方法，方便控件抓取时使用）。

配置好环境后的Libraries情况如图5-6所示。

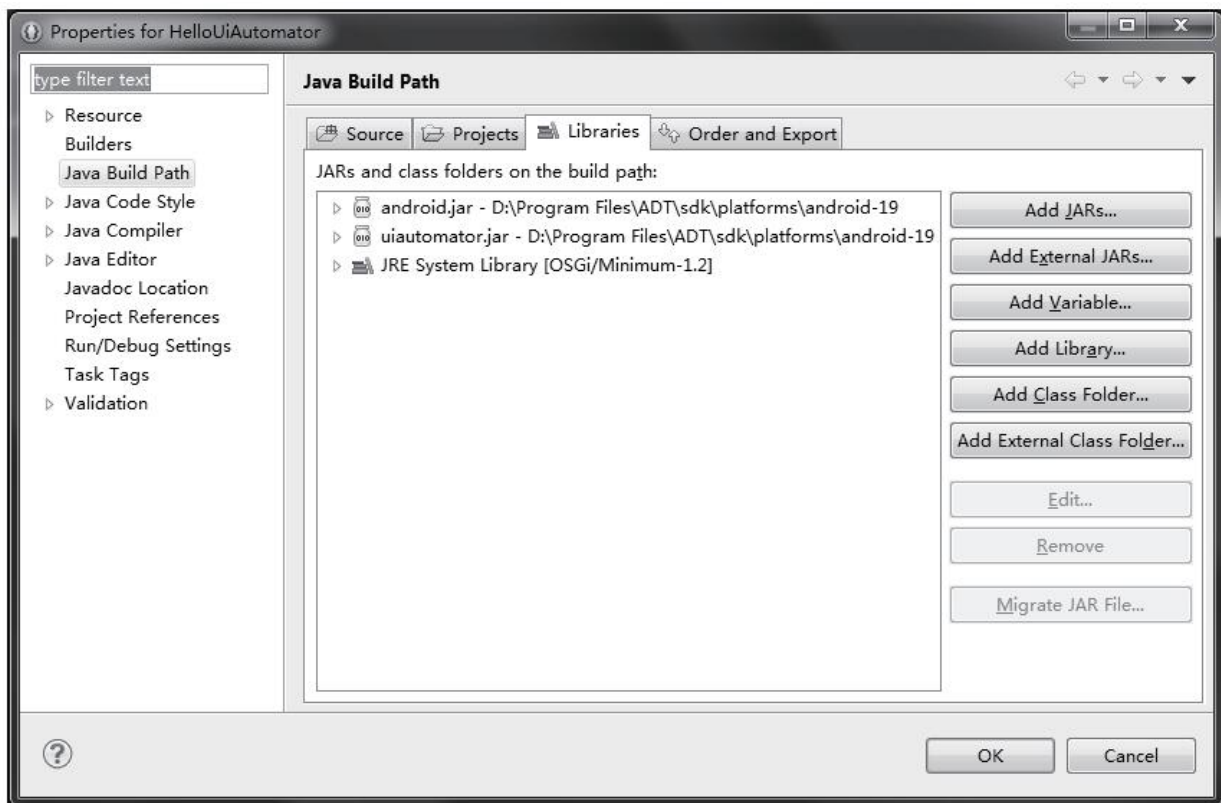


图5-6 配置好环境后的Libraries情况

3.测试用例编写

我们以测试系统设置自动休眠时间策略是否有效为例，写一个测试用例来完成自动化测试。页面的切换操作顺序如图5-7所示，先打开系统设置页面，然后进入显示页面，再单击休眠按钮，选择相应的休眠时间来完成设置。对于UIAutomator的用例编写，有几个基础规范如下：

- 每个用例都继承于UiAutomatorTestCase，可见域为public。
- setUp () 于每个测试用例前执行，适于用作环境准备。
- tearDown () 于每个测试用例后执行，适用于数据收集及环境恢复。
- 一个类可以有多个测试用例，用例之间不建议耦合，执行期间为无序执行。



图5-7 系统设置自动休眠时间用例 页面跳转

依据上述的测试顺序，编码得到测试用例如代码清单5-10所示。

代码清单5-10 系统自动休眠策略测试用例

```
public class HelloUiAutomator extends UiAutomatorTestCase {
    private static final String TAG = HelloUiAutomator.class.getSimpleName();
    private static final String FORMAT_LOG = ">>> %s [%s] %s";
    private static final SimpleDateFormat DATE_FORMAT = new SimpleDateFormat(
        "yyyy-MM-DD hh:mm:ss");
    private static final long TIME_OUT_FOR_EXISTS = 5 * 1000L;
    protected void setUp() throws Exception {
        log(TAG, "setUp of " + getName());
        super.setUp();
    }
    /**
     * 测试场景：验证自动休眠设置是否有效

<br>
     * 操作过程:

<br>
     * > 打开系统设置，设置自动休眠时间为
30S<br>
```

* > 休眠

30S. 检查屏幕状态

* 预期结果: 屏幕为灭屏状态

```
*/
public void testSetScreenOffTime() throws IOException,
    UiObjectNotFoundException, RemoteException {
    // 打开系统设置页面

    Runtime.getRuntime().exec("monkey -p com.android.settings -v 1");
    // 找到滚动列表: 这里以包名、
```

resourceId作为条件表示系统设置滚动列表

```
UiScrollable mSettingList = new UiScrollable(new UiSelector()
    .packageName("com.android.settings")
    .resourceId("android:id/list"));
mSettingList.waitForExists(TIME_OUT_FOR_EXISTS);
assertTrue("System setting list not exists", mSettingList.exists());
// 获取显示控件并点击进入显示设置: 使用
```

UiScrollable获取子控件会自动滚动寻找

```
log(TAG, "Enter display setting page");
UiObject mDisplayEntry = mSettingList.getChildByText(
    new UiSelector().resourceIdMatches(".*title"), "显示
");
assertTrue("Display setting entry not exists",
    mDisplayEntry.exists() && mDisplayEntry.isEnabled());
mDisplayEntry.clickAndWaitForNewWindow();
// 单击休眠按钮进入设置

log(TAG, "Enter sleep timeout setting page");
UiObject mSleepTimeEntry = new UiObject(
new UiSelector().textContains("休眠
"));
assertTrue("Sleeping timeout setting entry not exists",
    mSleepTimeEntry.exists() && mSleepTimeEntry.isEnabled());
mSleepTimeEntry.clickAndWaitForNewWindow();
// 单击设置为
```

30S超时

```
log(TAG, "Set sleeping timeout to 30s");
UiObject mTargetTimeOut = new UiObject(
new UiSelector().textMatches("30 秒
|30s"));
assertTrue("Can't find target timeout for setting",
    mTargetTimeOut.exists() && mTargetTimeOut.isEnabled());
mTargetTimeOut.clickAndWaitForNewWindow();
```



```

        // 检验结果

        log(TAG, "Sleep 30s for checking auto sleep");
        UiDevice mUiDevice = getUiDevice();
        SystemClock.sleep(30 * 1000);
        assertFalse("Auto sleep didn't work", mUiDevice.isScreenOn());
    }
    protected void tearDown() throws Exception {
        log(TAG, "tearDown of " + getName());
        super.tearDown();
    }
    /**
     * 返回当前系统时间

     * @return yyyy-HH-DD hh:mm:ss
     */
    private static String getCurTime() {
        return DATE_FORMAT.format(new Date(System.currentTimeMillis()));
    }
    /**
     * 输出日志

     * @param tag TAG
     * @param message 消息

     */
    private static void log(String tag, String message){
        System.out.println(String.format(
            FORMAT_LOG, getCurTime(), tag, message));
    }
}

```

4.测试项目编译

编写完测试用例后，我们需要将代码编译成jar包以供测试使用，UIAutomator自动化测试项目编译使用的是ant（由Apache提供的将软件编译、测试、部署等步骤联系在一起加以自动化的一个工具，ADT Bundle环境中已经集成，使用Eclipse环境可以移步<http://ant.apache.org/>下载）编译方式。我们此前创建的只是普通的Java项目，所以在首次编

译的时候需要为项目添加ant编译脚本，在Android指令中已经自带了这方面的支持，指令如下面的代码所示。

```
> android list
  id: 12 or "android-19"
  Name: Android 4.4
  Type: Platform
  API level: 19
  Revision: 1
  Skins: HVGA, QVGA, WQVGA400, WQVGA432, WSVGA, WVGA800 (default)
  ABIs : armeabi-v7a
  .....
> android create uitest-project -n projectName -t targetID -p projectPath
```

·targetID: 编译使用的Android Level在本机上对应的ID，可使用android list查看。

·projectName: 编译目标项目名称，也是对应生成的jar包名称。

·projectPath: 编译目标项目根目录路径。

执行完以上指令后，在项目路径下会新增三个文件，如图5-8所示。

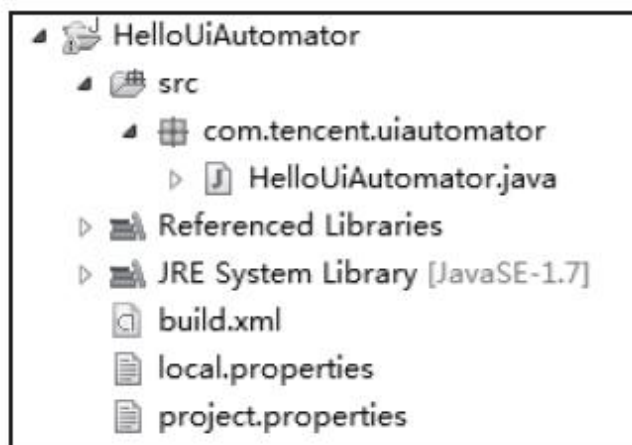


图5-8 生成UIAutomator项目后工程目录结构

上图所示的文件，具体说明如下：

- build.xml**: ant编译脚本，为/Android SDK/tools/ant/build.xml的副本。

- local.properties**: 存储本机SDK路径，若SDK目录迁移可在此做对应修改。

- project.properties**: 存储编译使用的API Level，如target=android-19。

至此，可以直接使用ant对项目执行build的操作（在build.xml文件上单击右键→Run As→Ant Build，或者在终端cd至build.xml所在目录，执行ant build指令），完成后在项目的bin目录下即可以看到以项目名称命名的jar包，即为最终测试使用的产物。

5.测试用例执行

完成项目编译后，就可以连接设备进行测试了，过程分为两步：先将所需的jar包推送至目标/data/local/tmp目录下，然后指定要测试的用例开始执行测试过程。

```
>adb push HelloUiAutomator.jar /data/local/tmp
378 KB/s (2716 bytes in 0.007s)
>adb shell uiautomator runtest HelloUiAutomator.jar
```

```

INSTRUMENTATION_STATUS: numtests=1
INSTRUMENTATION_STATUS: stream=
com.tencent.uiautomator.HelloUiAutomator:
INSTRUMENTATION_STATUS: id=UiAutomatorTestRunner
INSTRUMENTATION_STATUS: test=testSetScreenOffTime
INSTRUMENTATION_STATUS: class=com.tencent.uiautomator.HelloUiAutomator
INSTRUMENTATION_STATUS: current=1
INSTRUMENTATION_STATUS_CODE: 1
>>> 2015-11-24 11:48:39 [HelloUiAutomator] setUp of testSetScreenOffTime
>>> 2015-11-24 11:48:41 [HelloUiAutomator] Enter display setting page
>>> 2015-11-24 11:48:43 [HelloUiAutomator] Enter sleep timeout setting page
>>> 2015-11-24 11:48:46 [HelloUiAutomator] Set sleeping timeout to 30s
>>> 2015-11-24 11:48:50 [HelloUiAutomator] Sleep 30s for checking auto sleep
>>> 2015-11-24 11:49:20 [HelloUiAutomator] tearDown of testSetScreenOffTime
INSTRUMENTATION_STATUS: numtests=1
INSTRUMENTATION_STATUS: stream=.
INSTRUMENTATION_STATUS: id=UiAutomatorTestRunner
INSTRUMENTATION_STATUS: test=testSetScreenOffTime
INSTRUMENTATION_STATUS: class=com.tencent.uiautomator.HelloUiAutomator
INSTRUMENTATION_STATUS: current=1
INSTRUMENTATION_STATUS_CODE: 0
INSTRUMENTATION_STATUS: stream=
Test results for WatcherResultPrinter=.
Time: 40.75
OK (1 test)
INSTRUMENTATION_STATUS_CODE: -1

```

在执行过程中，会有instrumentation及我们自定义的Log进行输出，按执行流程输出的顺序为：执行前后instrumentaion输出执行状态信息（使用IautomationSupport输出的日志也会在此展示），中间夹着执行过程中我们输出的日志。对于日志信息的几个字段，信息意义如下：

- numtests=1：本次用例执行的总用例数为1。
- id=UiAutomatorTestRunner：本次执行者的ID为UiAutomatorTestRunner。
- test=testSetScreenOffTime：当前执行用例方法名为testSetScreenOffTime。

·`class=com.tencent.uiautomator.HelloUiAutomator`: 当前执行用例的类名为`com.tencent.uiautomator.HelloUiAutomator`。

·`current=1`: 当前执行用例的序号为1。

·`Test results for WatcherResultPrinter=.`: 本次总的执行结果，对应用例顺序，“.”表示用例执行成功，“F”表示用例执行失败，“E”表示执行用例出错。

5.3.2 UIAutomator设计思想

UIAutomator框架本身比较简单，留给开发人员自由发挥的余地也比较大。我们可以根据自身的需求，对其进行二次封装改造，但其自身的某些用例设计思想，还是比较优秀、实用的，建议在后续的开发过程中继续保留。

1.断言中断式用例设计

UIAutomator用例继承于`junit.framework.Assert`，在测试过程中可以使用断言对检查点进行校验。实际上Junit的用例设计思想也是这样的，默认用例顺序执行完毕表示用例执行成功，在执行过程中有异常抛出则认为用例执行失败。对于测试用例的设计来讲，就需要在关键的检查点插入校验状态的断言，一旦发现与预期逻辑不符，用例执行必定失败，则抛出异常，中断当前用例的执行。

中断式的用例设计思想与顺序式的写法有什么不同呢？我们通过上面检验休眠设置的例子来大体描述一下两种思想的代码写法。对于用例中描述的场景为：打开设置→单击显示按钮→单击休眠按钮→单击30秒按钮，在这个过程中，我们需要确保这三个单击的按钮一定存在，否则用例肯定会执行失败。

顺序式的写法如代码清单5-11所示，可以看到每个check Point都加入了一个if分支判断，代码整体上看比较冗长，嵌套较多，便于理解但不便于阅读及后期维护。

代码清单5-11 顺序式写法代码

```
if(mSettingList.exists()){
    mDisplayEntry = mSettingList.getChildByText("显示
");
    if(mDisplayEntry.exists()){
        mDisplayEntry.clickAndWaitForNewWindow();
        if(mSleepTimeEntry.exists()){
            mSleepTimeEntry.clickAndWaitForNewWindow();
            if(mTargetTimeOut.exists()){
                mTargetTimeOut.click();
                SystemClock.sleep(3 * 1000);
                if(!mUiDevice.isScreenOn())
                    System.out.println("Test success");
                else throw new Exception("Device is screen on");
            }
            else{
                throw new Exception("TargetTimeout not found");
            }
        }
        else{
            throw new Exception("SleepTimeEntry not found");
        }
    }
    else{
        throw new Exception("DisplayEntry not found");
    }
}
else throw new Exception("SettingList not found");
```

再来看中断式的代码，风格完全不一样，所有的if判断分支都采用断言进行替换，不用显示抛出异常，没有代码嵌套，干净、舒适，便于阅读，操作步骤之间可以用段落分开，整体代码逻辑比较紧凑，方便后续代码维护，如代码清单5-12所示。

代码清单5-12 中断式写法代码

```
assertTrue("SettingList not found", mSettingList.isExists());
mDisplayEntry = mSettingList.getChildByText("显示
");
assertTrue("DisplayEntry not found", mDisplayEntry.isExists());
mDisplayEntry.clickAndWaitForNewWindow();
assertTrue("SleepTimeEntry not found", mSleepTimeEntry.isExists());
mSleepTimeEntry.clickAndWaitForNewWindow();
assertTrue("TargetTimeOut not found", mTargetTimeOut.isExists());
mTargetTimeOut.click();
SystemClock.sleep(3 * 1000);
assertFalse("Device is screen on", mUiDevice.isScreenOn());
```

2.setUp、tearDown

UIAutomator用例继承于junit.framework.TestCase，其中有两个方法：setUp和tearDown。从字面上来看非常好理解，在每个测试方法执行之前都先执行setUp，在每个测试方法执行之后执行tearDown，比如测试类中存在着方法testA（）、testB（），则执行的顺序是setUp（）→testA（）→tearDown（）→setUp（）→testB（）→tearDown（）。

setUp通常用于做测试用例前的环境准备及状态记录，如开启LOG、关闭Wi-Fi等。对于上面的路径大家可能会存在这样的疑问，由于同一个类中的所有测试用例执行的都是同一个setUp方法，那么若不同的用例需要不同的前置条件，有没有什么办法满足呢？答案是肯定的。一方面，我们可以在setUp方法中通过getName（）来获取当前执行用例的名称，以此来区分不同的用例需要执行的前置操作；另一方

面，建议把用例按模块进行分类，同一类中的用例前置条件保持基本一致，部分不同的细节，也可以放在具体的用例里进行调节。

对于同一测试类中的不同方法，若测试的环境准备只需要做一次，也可以在类中增加标志符，配合`setUp`只做一次初始化操作，比如添加`boolean hasInit`初始化为`false`，仅当`hasInit`为`false`时才执行`init`，并在执行完成后将其置为`true`，以此实现类前初始化操作。

`tearDown`方法通常用于做测试执行后的结果收集及环境恢复。通常结果的收集需要放在用例执行的最后，但在执行用例的过程中，可能会由于断言失败或者执行异常抛出等中断了当前的测试，那么结果收集的代码便不能执行了，这时`tearDown`能显示出非常大的优势，即不管用例执行中断与否，最后都会执行`tearDown`，类似于`finally`的用法，所以非常适用于结果收集、资源释放及环境恢复的操作。



注意 `setUp`、`tearDown`会于同一个类中所有用例前后均执行一次，而非只执行一次，所以一般同一场景的`case`可以归为同一类，方便测试环境设置及恢复。若`case`条件不同，则可以使用`getName`方法加以区分，以做不同的用例前置操作及场景恢复。

3.非耦合式用例设计

在实际的测试设计过程中，很有可能出现用例对环境依赖的冲突，比如A用例需要设置系统锁屏为无，B用例需要设置系统锁屏为密码解锁，倘若初始环境下锁屏设置为滑动解锁，则可以通过以下两种方案设计实现。

第一种方案是全面考虑所有可能并在setUp中解决，只关注环境初始化，不考虑场景恢复。在此方案中，用例的设计便与用例的执行顺序产生了关联，依据不同的顺序，A与B在setUp中所需要做的初始化工作便有了不同。

·A → B: A，设置锁屏为无；B，设置为密码解锁。

·B → A: B，设置为密码解锁；A，解锁密码，再设置锁屏为无。

出于以上考虑，A在用例设计时就不得不把两种执行顺序都考虑到了，在A初始化时检测当前是否有密码，若有，解除后再设置为无锁屏，这对于A来讲非常痛苦，因为A无法得知B设置的密码是什么。B也需要做好兼容，因为它不知道在此之前锁屏是被设置为无锁屏，还是被设置成滑动解锁。况且，在以后的测试中，考虑到随着新用例的增加，可能会对环境做出这样或者那样的修改，每次用例新增后都需要对此前的用例再进行一次遍历，看是否需要修改初始化环境，维护的代价也会越来越大。

第二种方案是我们比较推崇的，所有测试用例以初始环境为基准，在setUp中完成初始化，并在tearDown中恢复初始环境。比如上文中的例子，则A、B被设计为：

·A： setUp中设置为无锁屏， tearDown中恢复为滑动锁屏。

·B： setUp中设置锁屏为密码解锁， tearDown中恢复为滑动锁屏。

这样一来，思路就清晰了很多，A、B都只需要考虑自己所做的更改，A不需要知道B设置的密码，B也不用管在此之前是无锁屏还是滑动锁屏，所有的前置条件都是明确的，谁污染谁治理，谁改的谁负责改回去。按照这个规则，即便后续添加再多新用例，都不会对已存在的用例造成影响，代码的维护代价便可大幅度降低。

综上所述，在UIAutomator的用例设计过程中，建议所有测试用例之间都做到完全解耦，每一个用例都可以单独执行，需要按照一定顺序执行的、无法拆分的场景，建议都写成一个测试用例。

5.3.3 UIAutomator实践案例

行文至此，读者对于UIAutomator的了解也就比较全面了，对于后续自动化过程中的应用及拓展的理解，相信也都不在话下。余下的，就是需要一些时间，在实际的应用过程中熟悉对它的应用，并掌握属于自己的风格及技巧，更好地去发挥它的作用。

在自动化测试的领域里，其目的往往在于解放测试人力，抑或形成监控对异常进行警告。笔者作为TOS（Tencent OS）的一名测试人员，也是在实际的测试过程中，结识并慢慢深入了解UIAutomator的方方面面的。TOS作为2015年年初新出现的Android ROM，经验尚浅，需要进行的测试任务非常繁杂。在一个ROM的层面上，UIAutomator对于一些问题痛点能带来怎样的解决方案，从而产生更多的收益呢？在接下来的内容里，不妨来了解一下UIAutomator在TOS的日常测试过程中的应用。

2015年3月3日，TOS终于与用户见面了，开启了内测的征程。由于TOS用心的设计及不错的评测，论坛上开始有未适配机型用户的声音，渴望能体验TOS。为了让“小鲸”（TOS Logo形象，后文借指TOS）早日与更多的用户见面，新机型适配计划随之也被提上议程。但作为新起项目，人力及测试建设都存在很多不足，在没有人力新增的情况下适

配更多的机型，工作量直接翻番，怎么能挤出更多的时间来完成新机型的适配呢？

那时候大体的测试都还依靠手工进行，在TOS中大部分系统App都是经过深层定制的，为了保证发布的质量，测试流程不可删减，于是问题便从这块硬骨头可不可以不啃，转换为可不可以不由测试人员来啃了。在了解了当前主流的自动化测试框架后，经过对比，最终挑选了UIAutomator作为自动化技术选型，TOS作为Android OS，测试关注不同于App，需要对多应用进行跨进程的场景测试，在这一点上UIAutomator的优势非常明显。

1.TOS单元测试

磨好了刀，接下来就要看往哪里砍了。在TOS流程中，每次上线前都需要对适配机型进行基础用例的覆盖测试，以确保发版质量，花费人力在8人/天以上，而且这部分的测试用例较为核心，功能基础，一般路径都不会很深。如果可以做自动化支持，晚上下班回家前让自动化“跑”起来，第二天早上上班时就能看到测试结果了，节约的不仅仅是人力，更缩减了发布的流程。

想想都有点儿小激动，于是便开始筛选用例，设计用例，编码实现（偏业务功能走查，部分UI检查需要涉及截图对比）。按梳理完的结果，很快，用例也写得差不多了，在自己的PC上，就可以看到自动

化执行日志及结果了，心里甚是欢喜，那时候的测试结果如下面的代码所示。

```
.....  
  
INSTRUMENTATION_STATUS: current=71  
INSTRUMENTATION_STATUS: id=UiAutomatorTestRunner  
INSTRUMENTATION_STATUS: class=com.tencent.cases.SystemSettingTestCase  
INSTRUMENTATION_STATUS: stream=.  
INSTRUMENTATION_STATUS: numtests=71  
INSTRUMENTATION_STATUS: test=testWifiGetDetail  
INSTRUMENTATION_STATUS_CODE: 0  
INSTRUMENTATION_STATUS: stream=  
Test results for WatcherResultPrinter=.....  
F.....E....F.....F..F.....F.F.....F....
```

很快，问题也接踵而来。随着实现用例数量的增多，每次的测试结果日志变得越来越长，每次看执行结果，对应错误用例并定位原因都需要花费大量时间。况且，总不能把这样的执行结果填写在测试结果邮件里，笔者觉得还需要更好看的结果分析样式。

了解到UIAutomator是基于Junit进行编写的，笔者在尝试进行结果格式化之前便先到网上了解是否有较好的解决方案，后来也证实了这个寻找并没有白费，github上有一个项目automator-log-coverter（<https://github.com/dpreussler/automator-log-converter>）其所做的工作就是将UIAutomator的日志转换为Junit日志，借助uiautomator2junit.jar，就可以轻松地执行结果里得到一份标准的Junit报告了，有了这份Junit报告，对于结果的处理显然就容易了很多。

用例有了，报告有了，距离自动执行并发送结果报告的愿景已经不远了。有人说行为上的懒惰是推动人类不断向前发展的源泉，对此，笔者不置是非，笔者只知道自己很懒，懒到甚至连花2.85卡路里点击鼠标运行一个脚本也不想干，而是想让UIAutomator自己执行发送结果，笔者只要坐在家里安静地看看邮件就好，为了实现这件事，笔者找到了Jenkins。

Jenkins集群在这里就不多介绍了，大多数的自动化持续集成都会用到它，Jenkins支持按设定的策略自动执行测试Job，支持结果邮件发送，更支持插件扩展，其中对于Junit的日志处理有很多，这对笔者来说实在是再赞不过了。

关于Junit的结果处理，笔者选择了Junit Plugin，可以根据历史执行结果呈现自动化的结果趋势。为此，需要在Jenkins上每次将Junit结果进行归档（几百KB的xml，占用空间很小），添加完Junit Plugin以后，报告是这样的，浅色代表总的用例执行数，深色代表本次执行失败的用例数，可以清晰地看到历史执行趋势，如图5-9所示。

点击图表可以浏览执行结果详情页面，成功占比、失败的具体用例、失败日志及历史执行情况，都一目了然，如图5-10所示。使用Jenkins的Jelly邮件模板把图表及链接附上，后面的工作就是维护用例的稳定性及Job的执行策略了。

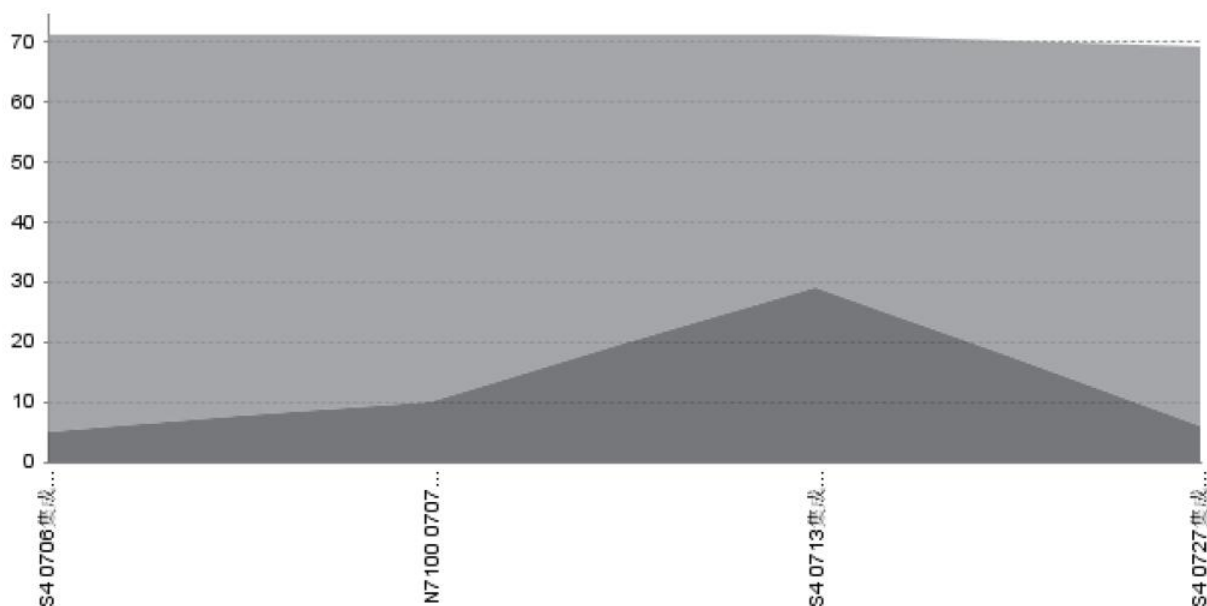


图5-9 应用Junit Plugin后UIAutomator执行结果的展示

Test Result

5次失败

71个测试
花了

添加说明

所有失败的测试

测试的名称	花的时间	寿命
com.tencent.uiautomator.cases.systemsetting.SystemSettingTestCase.testBlueCloseCanbeFound	0 毫秒	1
com.tencent.uiautomator.cases.systemsetting.SystemSettingTestCase.testSoundSetAlarm	0 毫秒	1
com.tencent.uiautomator.cases.systemsetting.SystemSettingTestCase.testAdvancedSettingStorageProcess	0 毫秒	4
com.tencent.uiautomator.cases.systemsetting.SystemSettingTestCase.testMobilenetOpen	0 毫秒	4
com.tencent.uiautomator.cases.systemsetting.SystemSettingTestCase.testMobilenetRoamOpen	0 毫秒	4

所有的测试

Package	花的时间	失败	(区别) 跳过	(区别) Pass	(区别) 总数	(区别)
com.tencent.uiautomator.cases.qweather	0 毫秒	0	0	13	+13	13 +13
com.tencent.uiautomator.cases.systemsetting	0 毫秒	5	+5	0	34	+34 39 +39
com.tencent.uiautomator.cases.systemui	0 毫秒	0	0	19	+19	19 +19

图5-10 Junit Plugin对某次执行结果的详情展示

2.TOS性能测试

解决了TOS的单元用例测试问题，想想后面的测试就可以听点儿音乐、喝杯咖啡，静静等待报告邮件发送了。UIAutomator这么好用的东西，怎么可以只停留在单元用例测试这里呢？


TOS虽然年轻，但有着一颗追求极致、不断挑战自我的心。为了整机更好的性能（App在指定场景下耗费系统资源的占比，如内存、CPU、FPS等），我们需要和资深MIUI等学习，通过性能测试对比来找到自己的“瓶颈”，寻找存在优化的空间。另外，我们也希望每周能有纵向对比，监控更新内容是否会对性能产生影响。TOS测试团队对齐了所有App的性能对比项，并将其固化下来作为ROM的核心能力，定期进行性能测试及竞品对比测试，以此来不断提升自己，同时形成监控，对性能影响较大的修改形成预警。

但性能的测试非常耗费时间，比如整体的灭屏待机率需要待机休眠24小时；内存及流畅度的一个场景测试，可能需要测试人员不停地滑动操作半个小时，而且期间存在着测试人员不同使结果产生差异的因素，比如滑动的快慢、前置条件存在差异等，所以经常需要多次测试以确定稳定值，仅TOS的性能测试往往就需要耗费15人/天。

在这里，UIAutomator强大的两个特性就得到体现了，一是测试不需要代码，对于竞品的测试拿过来一样可以跑起来；二是跨进程有强大的兼容性，性能的数据采集可以轻松接入第三方App进行，且不需要专门适配。

针对性能测试需求，笔者尝试接入了腾讯性能随身调工具GT（由腾讯研发用于移动端对App进行性能监控及调试的应用，详情可参见<http://gt.qq.com>），可以方便地采集目标App的内存、CPU、流畅度等方面的数据。接入的方式很简单，应用UIAutomator的跨进程操作能力，直接对GT进行界面操作即可，也可以将GT工具的操作流程封装成静态工具类，在需要的时候直接调用会更加方便。

关于静态工具的封装，工具类可以直接继承于UiAutomatorTestCase，方法声明为静态public即可使用UIAutomator的自动化操作交互。唯一的区别在于，在普通的测试用例里，可以调用getUiDevice（）获取UiDevice实例，但静态方法中无法调用getUiDevice（），所以需要使用UiDevice.getInstance（）来替代。

 **提示** UiDevice实例是在UIAutomator执行准备时注入的，使用UiDevice.getInstance（）或者在继承UiAutomatorTestCase实例中使用getUiDevice获取的是同一个实例，一个是类静态方法，另一个是实例方法，可以根据使用场景选择合适的方法进行调用。

至此，使用UIAutomator操作性能测试场景，使用GT记录性能数据，两者都实现了。但存在一个问题，调用GT记录性能数据的操作过程以及性能测试场景的前期准备期间，性能已经开始记录了，在这段时间内采集的数据不能作为性能计算范围，需要予以剔除，但作为GT，它无法知道性能测试场景开始的精确时间，那怎么办呢？

解决方案很简单。在自动化过程中，当性能场景开始复现和结束时，通过LOG输出两个时间点，通过IautomationSupport记录到结果日志中，结束后通过取得用例的时间点，到GT保存数据中根据时间戳进行数据截取，就可以得到精确的测试数据了。

再做一个简单的历史对比脚本，同样集成到Jenkins上，又可以自动下发执行，继续喝咖啡等邮件了，得到的邮件预览如图5-11所示，UIAutomator又可以继续为“小鲸”扬帆保驾护航了。

测试场景	上一版本 PSS (MB)	当前版本 PSS (MB)	同比增减	上一版本 CPU (%)	当前版本 CPU (%)	同比增减	上一版本 SM	当前版本 SM	同比增减
锁屏_静置场景_TOS	35.29 35.47	37.38 37.59	15.93%	6.43 36.36	3.98 29.11	1-38.15%	94.00	95.00	11.06%
锁屏_滑动场景_TOS	35.73 35.96	39.07 39.33	19.35%	22.23 26.63	23.64 28.37	16.34%	98.00	99.00	11.02%
桌面_前台静置场景_TOS	45.88 49.37	45.68 49.57	10.44%	10.43 30.88	3.88 19.42	162.80%	--	--	--
桌面_后台静置场景_TOS	48.38 51.75	40.99 41.83	1-15.26%	0.83 4.13	0.03 1.42	1-96.73%	--	--	--
桌面_桌面滑动场景_TOS	46.15 49.28	46.41 50.24	10.56%	14.64 23.21	16.67 26.78	113.87%	96.00	96.00	0.00%
桌面_文件夹操作场景_TOS	51.48 70.74	48.44 55.66	1-5.90%	10.53 24.11	10.22 28.86	1-3.00%	83.00	82.00	1-1.20%
桌面_前台静置场景_GO 桌面	66.93 67.26	69.39 69.53	13.67%	0.55 8.72	0.49 8.88	1-11.90%	--	--	--
桌面_后台静置场景_GO 桌面	66.22 66.52	68.43 68.63	13.33%	0.29 1.94	0.20 1.70	1-32.48%	--	--	--
桌面_桌面滑动场景_GO 桌面	55.36 78.36	56.74 72.27	12.49%	23.42 28.62	26.77 34.36	114.31%	92.00	95.00	13.26%
桌面_前台静置场景_360 桌面	35.06 36.10	35.99 36.51	12.65%	0.56 23.52	0.05 5.64	1-90.94%	--	--	--
桌面_后台静置场景_360 桌面	32.80 32.81	35.97 36.03	19.66%	0.02 2.38	0.01 2.23	1-32.34%	--	--	--
桌面_桌面滑动场景_360 桌面	34.98 36.50	37.07 39.25	15.97%	9.05 19.70	9.96 15.28	110.13%	96.00	96.00	0.00%
桌面_文件夹操作场景_360 桌面	60.71 61.84	71.30 72.00	117.45%	9.38 25.22	8.73 15.81	1-7.01%	81.00	85.00	14.94%
SystemUI_桌面静置场景_TOS	23.59 23.95	24.05 24.58	11.91%	1.66 9.16	0.18 4.91	1-88.98%	--	--	--
SystemUI_通知栏静置场景_TOS	23.75 24.35	22.95 24.30	1-3.39%	2.01 13.85	0.45 14.09	1-77.82%	--	--	--
SystemUI_快捷开关静置场景_TOS	23.68 23.94	23.04 23.47	1-2.72%	2.34 15.97	0.42 16.89	1-82.09%	--	--	--
SystemUI_通知栏滑动场景_TOS	24.01 24.43	23.43 23.62	1-2.43%	10.46 17.62	10.51 15.24	10.42%	84.00	83.00	1-1.19%
RecentUI_任务栏静置场景_TOS	27.20 30.90	29.42 30.84	18.14%	7.03 11.76	9.02 16.21	128.30%	95.00	94.00	1-1.05%
RecentUI_任务切换静置场景_TOS	27.20 30.90	27.61 29.06	11.51%	0.07 12.06	0.27 7.11	1285.71%	--	--	--
系统设置_打开静置场景_TOS	26.76 27.88	31.39 32.20	117.29%	3.44 9.39	3.42 10.15	1-0.80%	--	--	--

图5-11 TOS性能自动化测试结果邮件

3.TOS压力测试

作为一名测试人员，也许会有这样的经历：在测试过程中遭遇非必现问题、对用户反馈问题无法复现、部分压力测试场景过于耗时等。很多时候，由于问题的偶然性，在开发人员定位问题的过程中无法提供有效信息，导致问题被长期搁置，往往也会在尝试复现的过程中耗费大量的人力。

在TOS内测发布后，有部分用户反馈手机容易出现卡顿、操作无响应等现象。在用户的协助下，我们从dropbox中获取了部分日志，定位到手机端时出现了low memory的现象，即系统可用内存极低造成卡

顿，只可惜日志不够详细，无法定位到具体原因。在此之后，测试人员努力尝试了两天多，但仍未能成功复现问题。

为了尽快解决定位问题，我们想通过自动化加大测试强度来尝试复现问题，既然是lowmem场景，那么就在手机端安装大量的第三方App，通过第三方App的唤醒来消耗系统内存使系统达到资源紧张的状态，具体实现的过程如下：

- (1) 在手机端安装大量第三方App。
- (2) 使用`pm list packages-3`获得第三方App列表包名。
- (3) 通过包名列表不断循环调起App，并退至后台不杀死。
- (4) 过程中使用`dumpsys meminfo`对内存采样，`logcat`抓取系统日志。
- (5) 监控dropbox目录是否存在lowmem日志，一旦出现即为问题复现。

以上过程可以作为一条UIAutomator的测试用例，通过编码在用例中调用Runtime来执行指令，完成App操作及信息监控。通过夜间的自动化压测，在重复至2000多次的App操作时，我们稳定地复现了该问题场景，为开发提供了有效的问题现场以定位。

在此之后，我们很重视App的压力测试环节，产品上线前，都需要跑一遍压测，其中最为普通的方法，便是使用Monkey进行。遗憾的是，Monkey是完全随机的压力测试过程，没有任何的业务逻辑存在于其中，对于部分需要登录或是顺序操作的场景，便无法覆盖了，而且对于部分不存在Activity的App（如锁屏、系统界面等）会如现如下代码所示的错误。

```
255|root@cancro:/ # monkey -p com.android.keyguard -v 10000
monkey -p com.android.keyguard -v 10000
:Monkey: seed=1451195107020 count=10000
:AllowPackage: com.android.keyguard
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
** No activities found to run, monkey aborted.
```

怎么解决这个问题呢？作为Monkey，其主要的事件分类有点击事件、按钮事件、滑动事件等，而这些在UIAutomator中都可以很方便地实现。以UiDevice为入口，可以使用click、swipe等方法来产生屏幕交互事件，使用pressKeyCode来产生按钮事件，只要模拟Monkey把事件进行归类，分配好事件发生的概率，就可以得到定制版本的压力自动化脚本了。相较于Monkey，UIAutomator压力自动化脚本有着不少优势：

- 使用灵活，可以巧妙加入业务逻辑，如登录、顺序跳转等。
- 可以加入性能监控、日志监控，自定义所需信息。
- 测试不依赖Activity，范围涵盖界面及指令级别测试。

- 可限定界面测试，相较于Monkey指定包名范围可以更小。
- 可以跨进程测试，不局限于同一包名，较于Monkey范围又可以更大。

借助于UIAutomator的强大功能，在压力测试这一环节，我们少走了很多弯路。最初TOS开始适配智能手表的时候，由于硬件性能限制及App规范不同等因素，一些本来很小的问题被放大了，系统及App经常会出现不稳定现象，比如Crash、ANR、内存泄漏，都会产生较大的影响。在这种情况下，对App的要求就更为严格，压测也就更为重要。

很可惜，在这种情况下大部分的测试App由于未对手表进行适配，都无法使用，于是我们借助UIAutomator模拟Monkey对手表上的App进行了系列测试，并封装了部分关键性能采集工具（PssMonitor、CpuMonitor、FpsMonitor），全程监控App的性能表现。前期很快地暴露了App的性能及稳定性问题，快速完成了稳定收敛的过程，为之后的测试开发争取到更多时间，并将此作为此后的常规监控项，加入持续集成平台中。

4.遇到的问题与解决

1) 乱码及输入问题

在开发过程中常会遇到字符编码的问题，而且是一个比较让人头疼的问题，平时编码过程采用UTF-8简直是编程界的“传统美德”。在UIAutomator编码的过程中，也有几个地方会涉及中文编码：text文案匹配（如UiSelector.text、UiScrollable.getChildByText）、控制台LOG输出信息、控件文案输入（如UiObject.setText）。

对于text文案匹配，解决的方法很简单，把当前项目编码调整成UTF-8即可。有些IDE默认随系统设置工作空间编码为GBK、GB2312等，就会导致定义的字符串以中文编码的方式进行。在UIAutomator控件匹配过程中，被翻译成UTF-8后就会失去原来的意思，导致控件无法匹配，这也是有些用户会认为UIAutomator不支持中文的原因。

控制台LOG输出信息由System.out来完成，我们可以将其再次封装并指定编码格式来解决中文乱码问题，对应的调用从System.out切换到mPrintStream，代码如下：

```
PrintStream mPrintStream = new PrintStream(System.out, true, "GB2312");  
mPrintStream.println(...);
```

最后中文输入的问题表现为依赖于当前输入法，比如调用语句setText（“tong”）进行输入的时候，当前如果是拼音输入法，可能会得到“同”；如果是五笔输入法，可能会得到“释怀”，结果变得不可预料。

我们可以通过引入Utf7Ime来解决这个不稳定问题，因为UiObject.setText（String）只能接受ASCII码，在我们输入的过程中，先用Utf7Ime将编码的字符串编码成ASCII码，setText接受这些ASCII码后再通过Utf7Ime这个输入法解码成我们此前编码的字符串输出，以此来保持输入内容的一致性。具体方法如下：

（1）打包下载，下载地址为<https://github.com/sumio/uiautomator-unicode-input-helper>。

（2）导入其中的Utf7Ime，生成apk并安装设置成默认输入法。

（3）把Utf7ImeHelper.java导入自己的公用方法库，用于把字符串encode成ASCII码。

（4）在自动化过程中，可以使用如下面代码所示的方法来查询及设置系统默认输入法。

> Android 4.2 以前:

```
dumpsys input_method | grep mCurMethodId
ime set jp.jun_nama.test.utf7ime/.Utf7ImeService
```

> Android 4.2 (含) 以后:

```
settings get secure default_input_method
settings put secure default_input_method jp..
utf7ime/.Utf7ImeService
```

2) 引用第三方包编译问题

当对UIAutomator使用越来越熟练的时候，笔者希望能把结果处理也放到自动化中来。Java的扩展库非常丰富，例如在本次的实战过程中就使用了json的第三方包，但在编译的时候就发现ant会报错，原因是json的包没有找到。

ant是根据脚本build.xml进行构建的，其内容主要部分引入了uibuild.xml来完成真正的构建过程，这个文件同样也在/SDK/tools/ant/目录下，内容如代码清单5-13所示。

代码清单5-13 build.xml中引入uibuild.xml部分代码

```
<!-- Import the actual build file.
      To customize existing targets, there are two options:
      .....

      - customize to your needs.
-->
<import file="${sdk.dir}/tools/ant/uibuild.xml" />
```

查看uibuild.xml的内容，我们就可以在compile和-dex的过程中加入我们的依赖库以完成编译与打包，修改可以如代码清单5-14所示，添加依赖libs编译脚本，更多的用法还可以检索ant build.xml学习。

代码清单5-14 添加依赖libs编译脚本

```
<target name="compile" depends="-build-setup, -pre-compile">
  <javac encoding="${java.encoding}"
        source="${java.source}" target="${java.target}"
        .....

        verbose="${verbose}"
```

```
        fork="${need.javac.fork}">
        <src path="${source.absolute.dir}" />
        <compilerarg line="${java.compilerargs}" />
        <classpath>
            <fileset dir="${jar.libs.dir}" includes="*.jar"/>
        </classpath>
    </javac>
</target>
<target name="-post-compile"/>
<target name="-dex" depends="compile, -post-compile">
    <dex executable="${dx}"
        output="${intermediate.dex.file}"
        nolocals="@{nolocals}"
        verbose="${verbose}">
        <fileset dir="${jar.libs.dir}">
            <include name="json.jar"/>
        </fileset>
        <path path="${out.classes.absolute.dir}"/>
    </dex>
</target>
```

3) 长时间执行adb不稳定问题

我们知道，通过adb shell执行UIAutomator的时候，如果想中断测试只要结束shell就可以了，非常方便，不过这也为后面的自动化带来了问题。随着测试用例越来越多，加之有的压力测试场景时间偏长，一次自动化测试过程时长可能在8小时以上。如果在执行的过程中USB断开，那么测试便会中断，而在PC上，手机助手或者管家卫士重启或是占用adb端口的现象时而存在，就算不被重启或是占用，在Windows上长时间的adb连接还是会因为供电不稳定等因素而发生断开，所以往往几个小时的测试都会白费。

怎么解决这个问题呢？前面讲到UIAutomator执行过程中有一个参数——--nohup，我们可以在执行的指令后面加上该参数，把UIAutomator挂起，在设备后台运行，这样设备是否保持adb连接就没有

关系了。至于对UIAutomator执行状态的获取，我们可以写脚本，通过ps查看进程是否仍在运行，轮询至进程停止即为执行结束，再拉取执行结果。

UIAutomator脱离adb执行后扩展性更强，可以兼容更多的自动化场景，比如耗电自动化测试（要求USB在非连接状态下进行），或者是通过apk调用UIAutomator进行自动化测试，封装常用功能供用户使用，或者是测试结果db持久化，等等，就都可以信手拈来了。

5.4 UIAutomator总结

UIAutomator自动化框架很简单，使用起来却非常灵活，其留给开发者自己拓展的自由空间还是比较多的。对于存在问题的地方，我们大可以对源码进行跟踪解读，多究根多思考，总能找到满意的解决办法，在Android界面自动化过程中，还是非常推荐使用UIAutomator自动化框架的。对于自己实践过程中的一些体会，也可稍作总结以相互学习。

5.4.1 UIAutomator代码规范及建议

在自动化编码的过程中，常常需要反复修改，就是用例兼容性及稳定性的增强。从App或系统层面的测试，到对于不同机型的兼容，我们需要考虑不同的分辨率等对于脚本产生的影响。虽然UIAutomator是基于控件进行的自动化操作，但在进行过程中还是有一些细节可以帮助提升脚本的兼容性及稳定性的。

- 遵循UIAutomator用例设计模式**。在自动化过程中，使用Assert断言加入预设逻辑，灵活控制case执行，善用setUp、tearDown方法，执行过程会更加清晰可控，也便于后期代码维护。

- 用例解耦，共同维护初始测试环境**。用例之前避免耦合关系，防止乱序执行带来不可预期的影响，共同维护测试环境，确保每个用例执行前初始环境的一致性，避免用例间执行的影响。

- 少用绝对坐标，基于控件或相对坐标进行**。绝对坐标在不同机型适配时兼容性差，维护成本高，尽可能使用控件进行交互，可以使用控件的边框信息Rect，或者使用屏幕的宽高信息按比例计算相对坐标。

·**少用sleep，多使用框架事件等待**。sleep方法休眠时间固定，时间长了影响测试速度，时间短了失败概率又会上升。建议使用waitForExists、waitUntilGone、waitForIde、waitForWindowUpdate等方法替代，这一系列方法在超时时间内一旦条件达成就会继续往下执行，不会浪费测试时间，在到达超时后也会继续向下执行，比sleep更为灵活，可以完全替代sleep。

·**减少API依赖，多思考替代方法**。对于官方的API，不要百分之百的依赖，存在疑惑的时候深究其根因，然后更好地改造、使用它。比如longClick的长按时间固定，但用户可以使用相同起终点的swipe来模拟长按并自定义时长，比如4.4.2版本中的scrollForward方法可能会因起终点无法触摸而失效，而swipe系列方法不存在该限制等，多思考，办法总会有的。

5.4.2 UIAutomator技巧及封装

UIAutomator虽然作为界面自动化框架，但在实际自动化测试应用过程中，还有不少内容可以挖掘，突破界面的限制，我们可以看到更多，用得更加顺手。

1.巧用Runtime

UIAutomator有个先天性的缺点，就是执行时界面匹配问题，每一个动作都需要进行界面遍历，所以在测试的过程中操作难显迅速，加之UI测试各界面渲染及跳转需要消耗时间，加长了用例执行的耗时。对于每个用例有其明确的检查点，在去往检查点的路径上，如果操作对于检查点没有影响，可以考虑能否使用指令代替，以节省时间。

比如在TOS上想打开日期设置，检查自动确定时区是否启用，通过纯UI操作，需要经历以下几步：进入桌面→进入应用检索→打开系统设置→打开高级设置→打开日期和时间设置→检查时区开关是否开启，如图5-12所示。

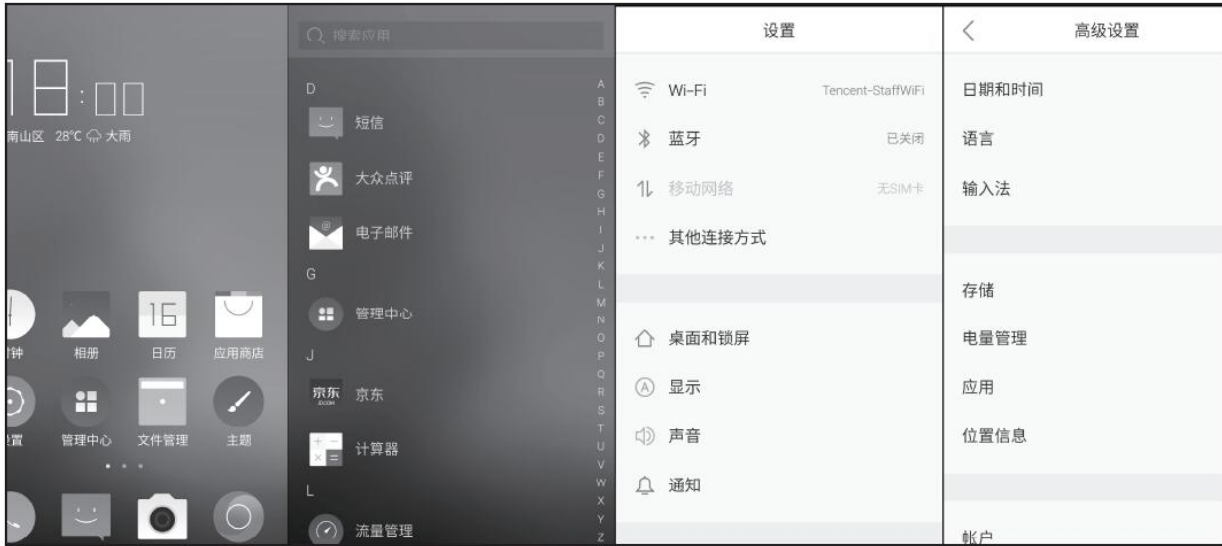


图5-12 确认自动时区功能是否打开UI操作过程

整个过程需要和五个页面交互，其中还包含三个滑动列表，编码实现需要至少八个类、几十行代码。但最终的检查点只存在于进入日期和时间设置页面之后开关的状态，中间怎么进入该页面对结果并没有影响，所以我们可以这样改一下：

通过Runtime执行指令打开日期和时间页面→检查时间同步开关是否开启。而前一步只需要一条指令：`am start-a android.settings.DATE_SETTINGS`，避开了繁杂的界面交互，且不存在不同机型的兼容性问题，用例执行时间可以大大缩短，同时兼容性、稳定性也有所提高，是非常巧妙的应用。

Runtime指令功能丰富，信息获取及结果处理非常灵活，我们可以借助它来获取系统信息、设置测试环境、封装性能监控工具等，结合

ROOT，操作系统几乎无所不能，也是自动化过程中的一大利器，非常推荐使用。

2.巧用settings

settings是Android 4.2之后引入的一个系统工具，其具体的作用是可以方便地获取设置设备变量，不需要ROOT权限，堪比一个比较BUG的存在，许多手机助手或者管家都利用它来获取额外的权限对设备进行操作，对于测试人员来讲，也有非常便利的地方。

首先来看看它的具体用法，可参见SettingsCmd.java源码中的帮助信息，settings指令的基本用法有两个，一是获取属性值（get），二是设置属性值（set），其指令帮助信息如下面的代码所示：

```
> settings
usage:  settings [--user NUM] get namespace key
        settings [--user NUM] put namespace key value
'namespace' is one of {system, secure, global}, case-insensitive
If '--user NUM' is not given, the operations are performed on the owner user.
```

namespace是名空间{system, secure, global}中的一个，至于具体的Key，可以参考官方API文档获取其含义。在这三个类里，都存在诸如getFloat、getInt、getString等方法。对于开发者来说，可能早已经不陌生了，其实这三个类取值的数据来源都是一样的，只是对settings.db内的值进行读取和修改而已，不同的名空间，只是对于Key的属性不同的归类而已。早期，大多数的值都是归属于system的，API Level 17之

后，部分system的Key被废弃，同时移至secure或global中，但Key的值保持不变，使用过程中需要注意API Level，如部分参数在一定Level后废弃，如图5-13所示。

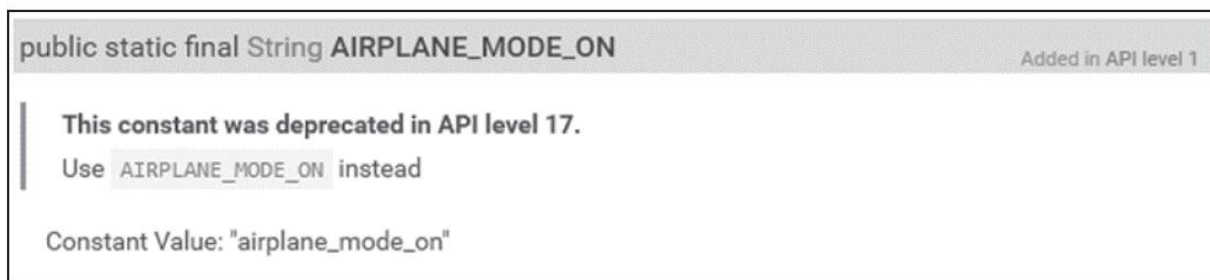


图5-13 API废弃后在官网添加的说明

对于settings中的值，大体有以下几种，也非常容易使用。

- enable or disable: 使用1、0的区分来表示enable、disable。
- mode: 使用int值来表示不同的模式，具体模式可以在常量定义中找到。
- String: 使用String来记录串值及URI。

作为Android的应用，在运行时需要与系统各种状态打交道，像Wi-Fi、蓝牙、定位等。在和这些模块交互的过程中需要申请相应的权限，Android的权限分类很多，使用起来并不是很方便。UIAutomator测试的jar包不能取得运行时的上下文，也无法配置相应的应用权限，而通过

settings类，我们却可以很方便地获取设备状态信息及设置相应的属性以供测试，如下面的代码所示。

```
settings get global auto_time           // 是否自动同步网络时间

settings get global wifi_on             // Wi-Fi是否打开

settings get system next_alarm_formatted // 获取下一次闹钟

settings put system screen_brightness 150 // 设置屏幕亮度

settings put system screen_off_timeout 600000 // 设置屏幕休眠时间
```



提示 2015年3月，Google把此前分散的测试组件合成了一个统一的Android Testing Support Library，UIAutomator由此被Instrumentation收入麾下，发布了2.0版本，这意味着UIAutomator也具备了Instrumentation的特性，可以获取Android应用上下文，也可以与Espresso等框架结合使用，功能变得更加强大。由于新版本使用方法及环境相差较大，在这里不展开详述，感兴趣的读者可以自行前往官网了解：<http://developer.android.com/intl/zh-cn/tools/testing/testing-tools.html>。

5.5 本章小结

UIAutomator作为一个基于控件不需要源码的自动化测试框架，其跨进程的支持及事件等待机制是极具优势的特性，站在测试的角度来讲，不管是什么工具，只要有利于提高测试效率及测试质量，就是好工具，UIAutomator在此表现出了强大的兼容性，可以方便地支持App类及指令类的功能拓展。其框架简单，容易上手，但功能强大，留给开发者自由发挥的空间很大，能满足绝大部分的测试需求。作为一名测试工程师，只要多尝试、多思考，我们相信解决的办法总会有的！

第6章 Appium框架解析及实践

本章将介绍与Appium测试框架相关的知识。如图6-1所示，在本章第一部分介绍了Appium框架的原理以及重要的技术特点，这些原理和特点可以在选择测试方案时帮助测试人员做决定。第二部分内容介绍了如何搭建Appium测试框架的环境和入手写一个基本的测试脚本，该部分知识对刚接触Appium框架的新手有一定的指导作用。第三部分是Appium应用的进阶知识，该部分内容介绍了Appium在腾讯地图这个项目中的实践过程，并对在实施测试过程中遇到的一些问题和解决方案进行了阐述。

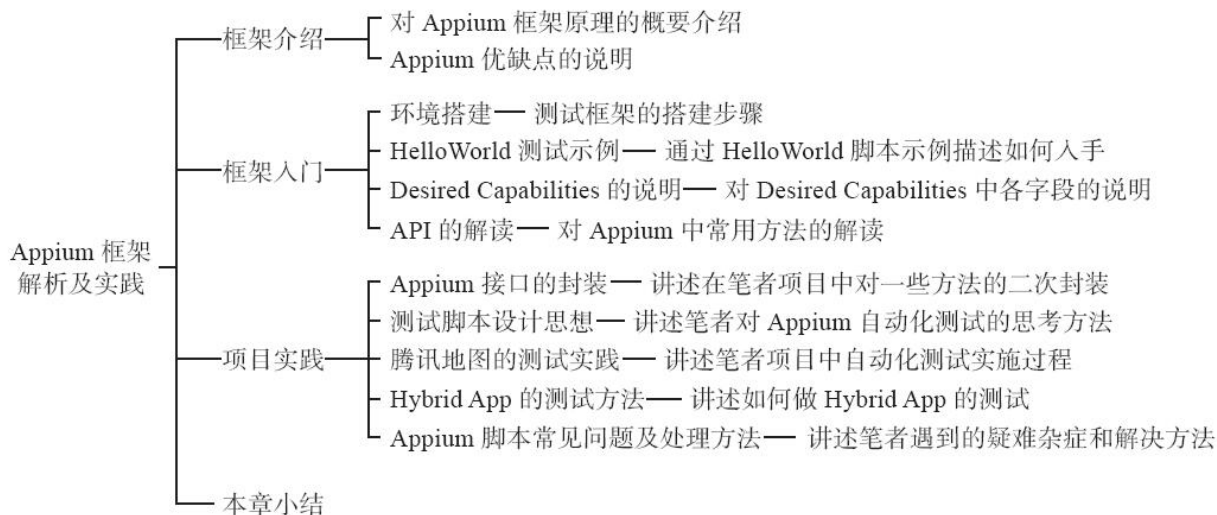


图6-1 本章知识结构图

由于目前Android 2.3的手机在市场上占比已经不足5%了，所以本章中的内容和示例是基于Android 4.4的操作系统，主要介绍Appium基

于UIAutomator进行自动化测试的知识，不详细介绍基于Selendroid的知识。此外关于Appium在iOS上的应用本章中不涉及，需要了解iOS相关知识的读者请通过其他方法查找。

6.1 Appium框架概况

Appium是一个开源的、跨平台的自动化测试框架，该框架适用于Native Application、Mobile Web Application或Hybrid Application的自动化测试。Native Application指的是基于智能手机本地操作系统如iOS和Android并使用原生编程语言（如Android上使用Java）编写并运行的第三方应用程序。Mobile Web Application指的是基于Web的系统和应用。Hybrid Application指的是在手机原生应用程序中嵌入了Webview，通过Webview可以访问网页的内容。

6.1.1 Appium架构原理

Appium是在手机操作系统自带的测试框架基础上实现的，Android和iOS的系统上使用的工具分别如下：

- Android（版本>4.2）：UIAutomator，Android 4.2之后系统自带的UI自动化测试工具。

- Android（版本≤4.2）：Selendroid，基于Android Instrumentation框架实现的自动化测试工具。

- iOS：UIAutomation，iOS系统自带的UI自动化测试工具。

Appium的架构原理如图6-2所示，由客户端（Appium Client）和服务器（Appium Server）两部分组成，客户端与服务器端通过JSON Wire Protocol进行通信。

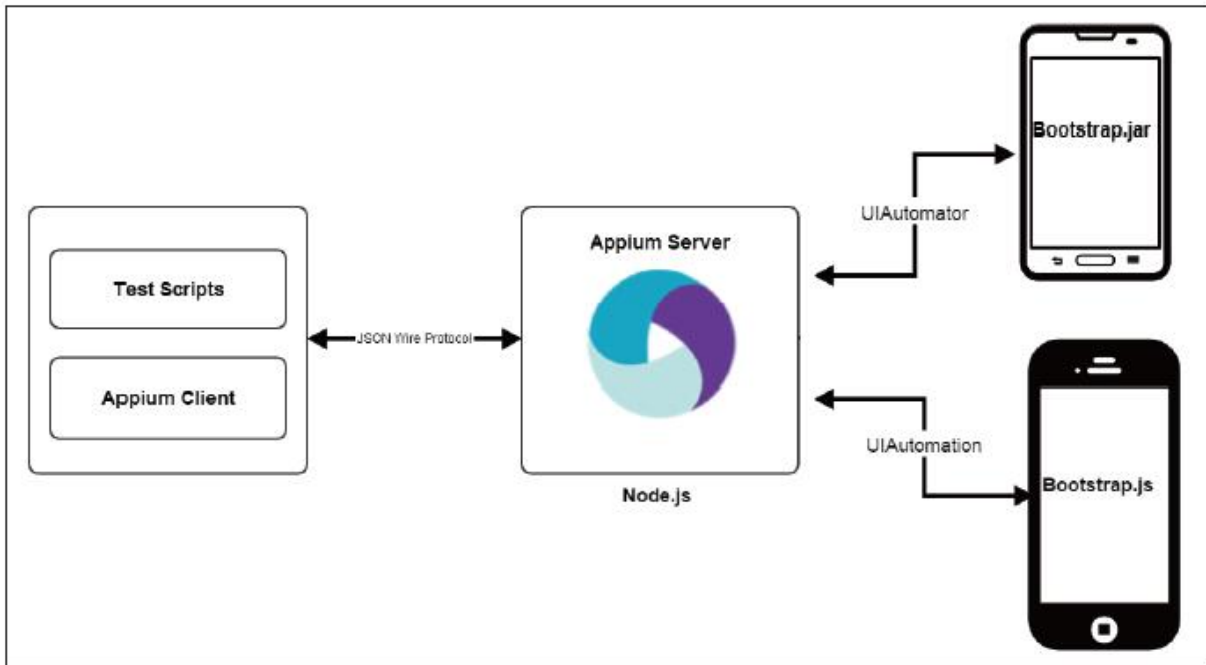


图6-2 Appium的架构原理

图6-2中各部分的作用及含义如下：

(1) Appium服务器。Appium服务器是Appium框架的核心。它是一个基于Node.js实现的HTTP服务器。Appium服务器的主要功能是接受从Appium客户端发起的连接，监听从客户端发送来的命令，将命令发送给bootstrap.jar（iOS手机为bootstrap.js）执行，并将命令的执行结果通过HTTP应答反馈给Appium客户端。

(2) Bootstrap.jar。Bootstrap.jar是在Android手机上运行的一个应用程序，它在手机上扮演TCP服务器的角色。当Appium服务器需要运行命令时，Appium服务器会与Bootstrap.jar建立TCP通信，并把命令发送给Bootstrap.jar；Bootstrap.jar负责运行测试命令。

(3) Appium客户端。它主要是指实现了Appium功能的WebDriver协议的客户端Library，它负责与Appium服务器建立连接，并将测试脚本的指令发送到Appium服务器。现有的客户端Library有多种语言的实现，包括Ruby、Python、Java、JavaScript (Node.js)、Object C、PHP和C#。Appium的测试是在这些Library的基础上进行开发的。

(4) Session。Appium的客户端和服务端之间进行通信都必须在一个Session的上下文中进行。客户端在发起通信的时候首先会发送一个叫作“Desired Capabilities”的JSON对象给服务器。服务器收到该数据后，会创建一个session并将session的ID返回到客户端。之后客户端可以用该session的ID发送后续的命令。

(5) Desired Capabilities。Desired Capabilities是一组设置的键值对的集合，其中键对应设置的名称，而值对应设置的值。Desired Capabilities主要用于通知Appium服务器建立需要的Session，其中一些设置可以在Appium运行过程中改变Appium服务器的运行行为。关于Desired Capabilities的内容将在6.2.3节详细阐述。

Appium在Android上基于UIAutomator实现了测试的代理程序(Bootstrap.jar)，在iOS上基于UIAutomation实现了测试的代理程序(Bootstrap.js)。当测试脚本运行时，每行WebDriver的脚本都将转换成Appium的指令发送给Appium服务器，而Appium服务器将测试指令交给代理程序，将由代理程序负责执行测试。比如脚本上的一个点击操

作，在Appium服务器上都是touch指令，当指令发送到Android系统上时，Android系统上的Bootstrap.jar将调用UIAutomator的方法实现点击操作；而当指令发送到iOS系统上时，iOS的Bootstrap.js将调用UIAutomation的方法实现点击操作。由于Appium有了这样的能力，同样的测试脚本可以实现跨平台运行。

6.1.2 Appium框架的优缺点

Appium框架的优点如下：

- Appium支持多种应用程序的测试。它可以测试移动Native App、Hybrid App和Web App。

- 被测试的应用程序不需要特殊编译。Appium的测试对象一般不需要做特殊修改，如不需要引入任何额外的测试SDK，不需要添加任何的权限，也不要求被测程序与脚本的签名一致，所以可以直接对发布的程序进行测试。但Hybrid App测试可能需要做一点儿修改，具体情况会在关于Hybrid App测试的6.3.4节中提到。

- Appium的脚本不限制语言和工具。由于Appium的客户端支持多种测试语言，测试人员可以选择熟悉的语言开发测试脚本，而其他的测试工具一般只能使用特定的语言。

- Appium支持应用之间跳转的测试。它可以用于测试多个应用程序相互交互的场景。

- Appium是一个跨平台的测试框架，可以使用同一个API开发出在Android和iOS上都可以运行的脚本。

Appium的缺点如下:

- 该工具必须连接电脑才能实施自动化测试，遇到需要脱机执行的场景就不能满足需求。

- 该工具只能用于UI的自动化测试，在很多情况下测试验证只能通过界面来进行。

6.2 Appium框架工作解析

从前一节内容中可以知道Appium测试框架在运行的时候需要搭建一个基于Node.js的服务器。本节选择以Window 7操作系统作为测试环境，以Python 2.7作为开发语言，测试手机选择LG Nexus 5（Android 4.4.4）搭建自动化测试环境，并且在该环境的基础上运行示例脚本。

6.2.1 Appium环境搭建

Appium环境搭建的具体步骤如下：

1.Android运行环境准备

在Android SDK安装结束后，右击“我的电脑”，选择“属性”→“高级系统设置”→“环境变量”，在系统变量里面，添加“ANDROID_HOME”的变量，并把Android SDK的安装路径设置为该变量的值，如图6-3所示。

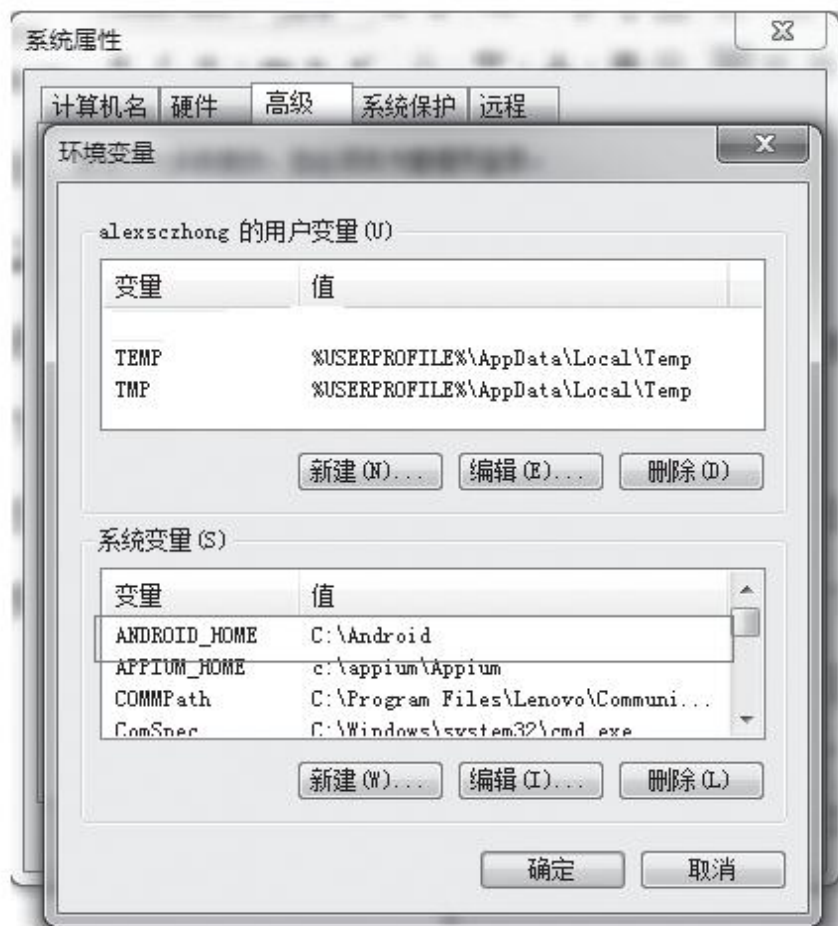



图6-3 Android SDK环境变量设置

2.安装Python

进入<https://www.python.org/downloads/> 页面下载最新Python2.7的安装包，然后安装。


参考图6-3中环境变量的设置方法，将Python的安装路径添加到系统变量的“PATH”中。

启动一个命令行窗口，输入“python-V”，如能正确显示版本号，则说明环境配置正确。

 **提示** 由于笔者使用的是Python2.7.11，本书后续的例子都将使用Python2.7的语法，如读者安装的是Python3.5，脚本语法上的差别请到Python官网查询。


3.安装Node.js

进入<https://nodejs.org/en/> 下载并安装Node.js。

 **提示** Node.js并不是必须安装的，由于在稍后的章节中会提到通过命令行的方式启动Appium服务器，这种方式需要安装一个独立的Node.js。

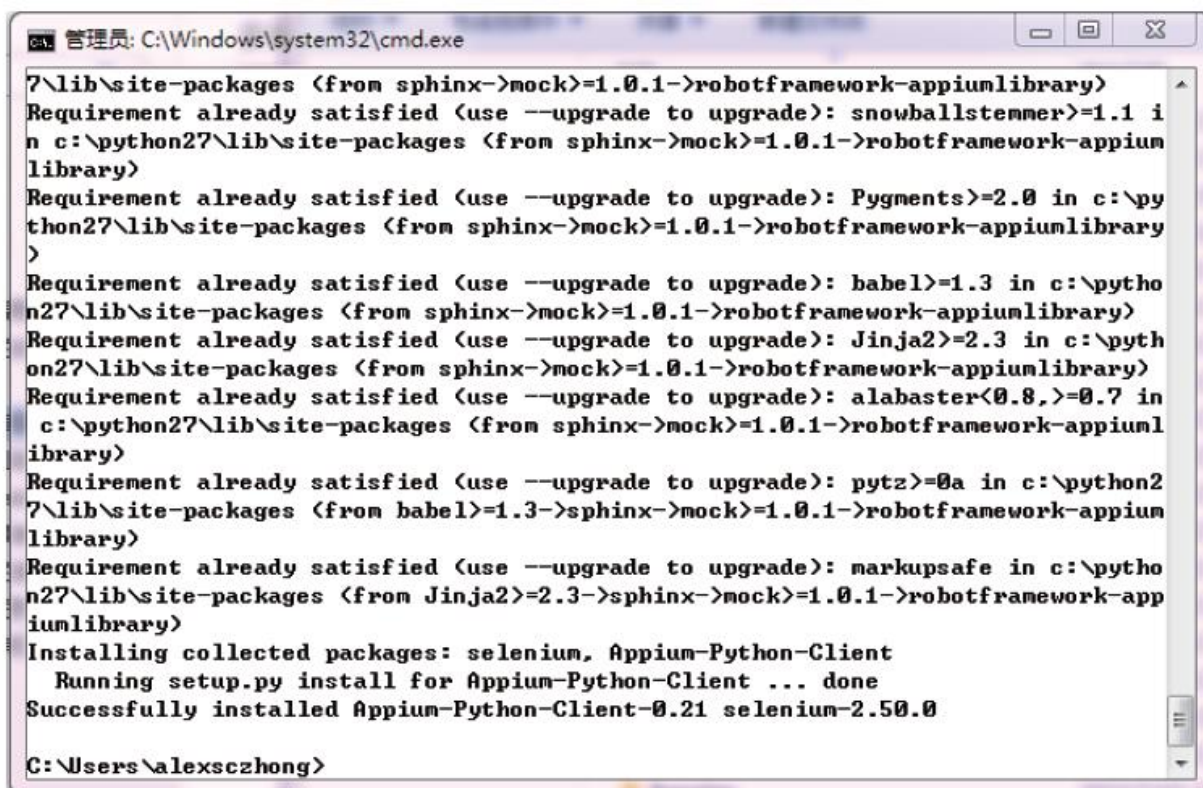
4.安装Appium服务器

进入<https://bitbucket.org/appium/appium.app/downloads/> 页面，下载Windows版本的Appium并安装。

 **注意** 建议读者安装Appium 1.4.0及以后的版本，笔者使用1.4.0之前的版本发现稳定性不好，程序运行过程中经常出现session建立失败，导致程序运行失败。

5. 安装Appium的客户端

启动一个命令行工具，输入“pip install robotframework-appiumlibrary”，然后按回车键，等待命令行提示安装成功，如图6-4所示。



```
管理员: C:\Windows\system32\cmd.exe
7\lib\site-packages <from sphinx->mock>=1.0.1->robotframework-appiumlibrary>
Requirement already satisfied (use --upgrade to upgrade): snowballstemmer>=1.1 i
n c:\python27\lib\site-packages <from sphinx->mock>=1.0.1->robotframework-appium
library>
Requirement already satisfied (use --upgrade to upgrade): Pygments>=2.0 in c:\py
thon27\lib\site-packages <from sphinx->mock>=1.0.1->robotframework-appiumlibrary
>
Requirement already satisfied (use --upgrade to upgrade): babel>=1.3 in c:\pytho
n27\lib\site-packages <from sphinx->mock>=1.0.1->robotframework-appiumlibrary>
Requirement already satisfied (use --upgrade to upgrade): Jinja2>=2.3 in c:\pyth
on27\lib\site-packages <from sphinx->mock>=1.0.1->robotframework-appiumlibrary>
Requirement already satisfied (use --upgrade to upgrade): alabaster<0.8,>=0.7 in
c:\python27\lib\site-packages <from sphinx->mock>=1.0.1->robotframework-appiuml
ibrary>
Requirement already satisfied (use --upgrade to upgrade): pytz>=0a in c:\python2
7\lib\site-packages <from babel>=1.3->sphinx->mock>=1.0.1->robotframework-appium
library>
Requirement already satisfied (use --upgrade to upgrade): markupsafe in c:\pytho
n27\lib\site-packages <from Jinja2>=2.3->sphinx->mock>=1.0.1->robotframework-app
iumlibrary>
Installing collected packages: selenium, Appium-Python-Client
Running setup.py install for Appium-Python-Client ... done
Successfully installed Appium-Python-Client-0.21 selenium-2.50.0

C:\Users\alexsczhong>
```

图6-4 安装Appium Client示意图

至此，Appium脚本运行所必需的程序就安装完毕了。读者可以尝试通过以下两种方式启动Appium服务器。

第一种方式是通过点击Appium安装路径下的appium.exe直接启动可视化界面，然后单击界面最右边的小火箭按钮，使用默认的配置启

动Appium服务器，启动画面如图6-5所示。

第二种方式是通过命令行启动。启动一个命令行窗口，将当前路径切换到Appium安装路径下的\node_modules\appium\bin目录下，输入命令“node appium.js--session-override”，单击回车键启动，启动画面如图6-6所示。

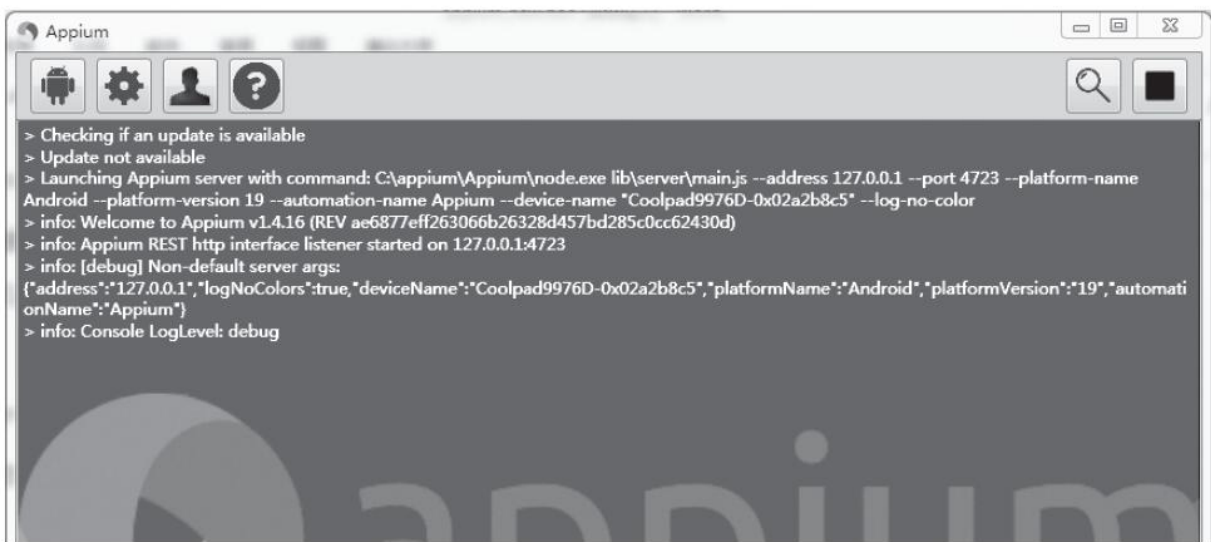


图6-5 通过UI界面启动Appium服务器

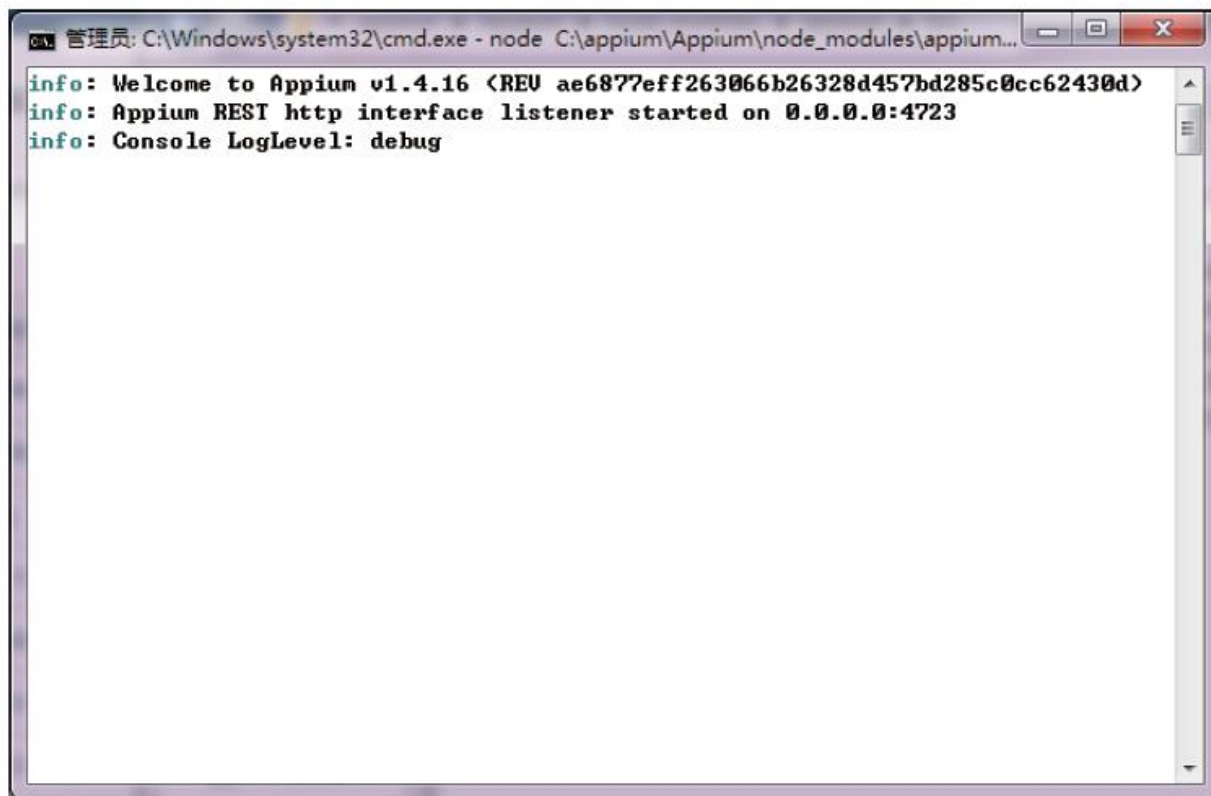



图6-6 通过命令行启动Appium服务器

 **提示** 第二种启动方式的“--session-override”参数非常重要，如果不设置该参数，那么当测试异常退出时，Appium服务器的session可能还存在，下一条用例尝试建立session就会失败。

笔者在实际项目中，更加倾向于使用第二种方法。由于自动化测试服务器都是部署在一台专用于运行自动化测试的机器上的，电脑重启后需要让Appium服务器自动启动，此时通过命令行的方式启动会更便捷。

6.2.2 HelloWorld测试示例

本节将在6.2.1搭建的测试环境的基础上，开发一个简单的测试脚本，该脚本将Android系统自带的电话本作为被测对象，通过自动化测试添加一条联系人。

首先我们需要创建一个脚本文件，并搭建出基本的用例结构，操作步骤如下：

步骤1：创建一个名为helloworld.py的Python脚本。

步骤2：通过Python的任意集成编辑器或者文本编辑器打开测试脚本，添加一个叫作“HelloWorld”的类，使该类继承自python的测试类unittest.TestCase。

步骤3：在该类中添加一个叫作“test_addContact”的方法。

步骤4：在测试脚本的后面添加文件的main入口，并在其之后通过unittest.TestLoader（）.loadTestsFromTestCase（）将“HelloWorld”类中的用例进行加载。此时helloworld.py文件的代码如代码清单6-1所示。

代码清单6-1 helloworld.py的示例

```
# -*- coding: utf-8 -*-
import sys
import os
```

```
import unittest
from time import sleep
class HelloWorld(unittest.TestCase):
    def test_addContact(self):
        pass
if __name__ == '__main__':
    suite = unittest.TestLoader().loadTestsFromTestCase(HelloWorld)
    unittest.TextTestRunner(verbosity=2).run(suite)
```

有了基本框架后，接下来就是往测试方法中添加Appium的测试脚本了。

测试脚本开发按以下步骤进行：

步骤1：在脚本开始位置处添加对Appium的Python客户端库的引入操作，需要添加的代码如下：

```
from appium import webdriver
```

步骤2：在“test_addContact”方法中添加一个字典变量，在本文中叫作desired_caps，向desired_caps添加如下面的代码示例所示的键与值。添加的每一个键值都对应Appium的Desired Capabilities中的一个设置。其中“appPackage”为被测对象的包名，“appActivity”的值为被测程序启动时的Activity。获取appPackage和appActivity的方法是将调试手机连接到电脑上，启动一个命令行窗口，键入命令“adb logcat>log.txt”并回车；然后手动单击被测应用程序的图标来启动应用程序；当程序启动后在命令行窗口按下“Ctrl+C”键结束日志，用文本编辑器打开log.txt文件，在日志文件中查找关键字“Start proc”，在该关键字后可以看到启动

的进程的appPackage和appActivity。appPackage的代码为“com.android.contacts”，appActivity的代码为“.activities.PeopleActivity”。

```
I/ActivityManager( 441): Start proc com.android.contacts for activity com.  
android.contacts/.activities.PeopleActivity: pid=3687 uid=10000 gids={50000,  
3003, 1015, 1028}
```

“deviceName”为连接的测试手机的名称，该名称可以通过命令“adb devices”获取到。本例使用的完整Desired Capabilities如代码清单6-2所示。

代码清单6-2 Desired Capabilities的示例

```
desired_caps = {}  
desired_caps['platformName'] = 'Android'  
desired_caps['platformVersion'] = '4.4.4'  
desired_caps['appPackage'] = 'com.android.contacts'  
desired_caps['appActivity'] = '.activities.PeopleActivity'  
desired_caps['deviceName'] = '052abd630a670a6a'
```

步骤3：以desired_caps作为参数初始化Webdriver连接，需要添加的代码如下面的代码所示。其中第一个参数是服务器的IP和端口组成的URL，本例中使用当前电脑运行Appium服务器，所以IP为127.0.0.1，而端口则可以在Appium服务器启动前指定，并且在Appium运行起来时会打印在日志里面内，详情请参考前文的图6-5和图6-6。当脚本运行到此处时可以看到Appium服务器开始建立，并且手机上会启动被测试应用程序。

#初始化

Appium连接

```
driver = webdriver.Remote("http://127.0.0.1:4723/wd/hub", desired_caps)
```

步骤4: 将测试手机连接到电脑上, 找到Android SDK安装目录下的tool\uiautomatorviewer.bat, 双击运行该脚本, 当UI Automator Viewer运行起来时, 将看到如图6-7所示的画面。单击图中所示工具条上的第二个按钮, 在UI Automator Viewer中可以看到当前手机上显示的画面。点击画面中某个控件时, 在右边的控件栏可以看到画面对应的控件树, 以及控件的信息。通过这样的方法就可以获取控件的信息, 这些信息将被用在测试脚本中。



图6-7 UI Automator Viewer运行示意图

步骤5: 将收集到的控件信息, 在测试脚本中调用Webdriver提供的方法先找到对应的控件, 然后操作控件。在本例中, 当电话本程序运行起来后, 首先是找到“创建新联系人”的按钮并单击, 示例的脚本如代码清单6-3所示。

代码清单6-3 “创建新联系人”按钮的代码示例

```
#查找创建新联系人按钮
```

```
createContactButton = None
```

```
try:
    #如果手机没有联系人，则通过
    create_contact_button来创建。

    此处通过控件
    id来查找

    createContactButton =
driver.find_element_by_id("com.android.contacts:id/create_contact_button")
except:
    #否则通过底部的添加联系人菜单来添加

    createContactButton =
driver.find_element_by_id("com.android.contacts:id/menu_add_contact")
    #单击创建按钮

    createContactButton.click()
```



提示

Webdriver提供的查找控件的方法有多种，包括通过控件ID、控件的文本、控件的类型和XPath等，但笔者建议读者首先尝试通过控件ID来查找。如果没有控件ID，笔者建议与开发者沟通让其添加。因为控件的ID一般很少改变，后期维护成本小；而控件的文本等信息变更的可能性很大，每次变动测试脚本都需要投入维护成本；另外，一个页面会有很多相同类型的控件，通过控件的类型不好定位。

步骤6：步骤5的操作执行完毕后，如果手机是第一次创建联系人，则将在页面显示出一个新的Dialog（图6-8），但如果不是第一次，则不会显示该Dialog。此处以显示该对话框为例，测试脚本在该步骤中应该用某种方法等待并判断该对话框显示后，再单击左边的按钮。如图6-8所示，选中Dialog后，找到该Dialog的一个FrameLayout的ID叫作“android: id/content”，示例先简单地通过sleep（）方法等待两

秒，然后查看该Dialog是否显示。接着单击左边按钮，如代码清单6-4所示。

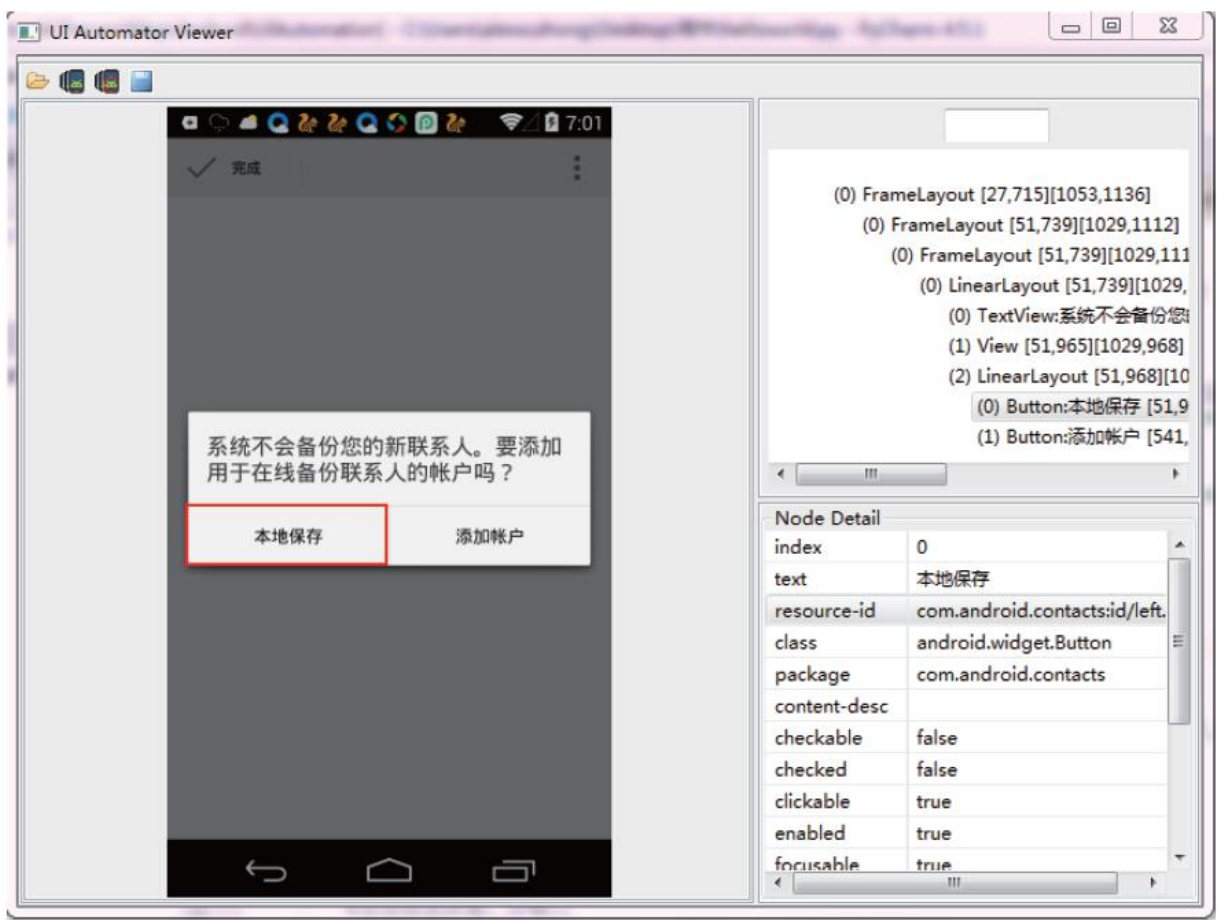


图6-8 弹出的新对话框

代码清单6-4 确认对话框显示时的处理代码示例

```
#稍等下，手机响应需要一点时间

#此处固定等待两秒方法不可取，由于不同的手机响应速度不同，脚本可能会失败

#此处仅是为了示例，在后面章节中会有更合理的等待方法

sleep(2)
#查看
```

Dialog的显示是否显示

```
try:
    dialog = driver.find_element_by_id("android:id/content")
    #找到“本地保存”按钮并点击

    saveLocal = driver.find_element_by_id(
        "com.android.contacts:id/left_button")
    saveLocal.click()
    sleep(2)
except:
    #如果找不到

Dialog或者

button, 就会跳转到这里

    print("no dialog found")
```

步骤7: 当上一步结束时, 屏幕会显示出一个新的Activity, 在这个Activity中将添加联系人的姓名和电话, 然后保存, 如代码清单6-5所示。在这里使用控件的文本来查找控件, 使用send_keys方法往文本框中输入文本, 最后截取的图片如图6-9所示。

代码清单6-5 添加联系人的代码示例

```
sleep(2)
#点击姓名, 并输入。此处是通过控件的文本来找到的

name = driver.find_element_by_name(u"姓名
")
name.click()
name.send_keys("appiumTest")
#点击电话输入框, 并输入。注意此处是通过找到一组控件, 并操作第
n个控件,
n从
0开始。

telephoneControls = driver.find_elements_by_name(u"电话
```

```
)  
telephoneControls[1].click()  
telephoneControls[1].send_keys("01012345678")  
#保存一个屏幕截图
```

```
driver.save_screenshot("afterinput.png")  
#单击完成按钮
```

```
completeButton = driver.find_element_by_id("com.android.contacts:id/icon")  
completeButton.click()
```



图6-9 添加联系人画面

步骤8: 在测试脚本最后添加验证的断言以及屏幕截图，本例中验证的内容是添加的联系人信息是否与预期输入的一致，并截图以备后

续人工检查。示例代码如代码清单6-6所示。

代码清单6-6 验证方法的示例代码

```
#验证添加的联系人信息是否与预期输入一样
```

```
barTitle = driver.find_element_by_id("android:id/action_bar_title")
self.assertEqual(barTitle.text, "appiumTest")
contactDatas = driver.find_elements_by_id("com.android.contacts:id/data")
self.assertEqual(contactDatas[0].text, "010 1234 5678")
#最后保存一个截图用于人工检查
```

```
driver.save_screenshot("newContact.png")
```

这样，测试脚本就开发完成了，将测试脚本保存。接下来按前一节描述的方式启动Appium服务器，另外启动一个命令行窗口，在命令行中切换到helloworld.py的保存目录下，运行“python helloworld.py”即可将测试脚本运行起来。



提示 完整的Helloworld测试脚本可以通过网址

http://tmq.qq.com/code_for_newbook/six.zip 下载得到。

以上测试脚本只是一个简单的示例，仅用于让刚接触Appium的读者知道如何入手去写一个Appium的测试脚本。该示例的稳定性不好，比如在该例子中直接固定等待两秒，手机的响应是不相同的，固定等待时间的方式不可取；另外，没有对测试脚本运行过程中出现异常的情况进行处理，如以上脚本中任意步骤出现异常，就可能导致整个脚

本运行中断，这些显然不是我们期望的。这些问题的解决方法将在6.3节的项目实践中进行详细阐述。

6.2.3 Desired Capabilities的说明

Desired Capabilities简单来说就是一组设置，这些设置可以让测试脚本控制Appium的运行行为。下面就逐个对Desired Capabilities中的设置进行阐述。

首先看与Appium服务器相关的Capability，表6-1中的Android和iOS两个平台都是有效的设置。

接下来是仅对Android测试有效的一些设置，见表6-2。

表6-1 与Appium服务器相关的Capability

Capability 名称	是否为必填项	描述	值
automationName	否	Appium 使用哪一个测试引擎，如果测试手机是 Android 并且在小于 API Level 17 的情况下，则 automationName 需要设置为 “Selendroid”	Appium (默认), 或者 Selendroid
platformName	是	被测设备是哪一种操作系统	iOS, Android 或 Firefox OS, null (默认)
platformVersion	否	手机系统版本	如 4.4.4, null (默认)
deviceName	否	使用的测试设备类型，该设置在测试 Android 时被忽略	例如: Nexus 5, null (默认)
App	否	指向 .apk 文件，或者包含有 .apk 的 ZIP 文件的本地路径或 http URL，Appium 在运行时会首先尝试安装 apk，然后开始测试。在测试 Android 时，如果设置了 appPackage 和 appActivity，则本 Capability 将被忽略	D:\tencentmap.apk 或 http://testserver/tencentmap.apk, null (默认)
browserName	否	手机网页测试时浏览器的名称。如果是测试手机应用程序，本 Capability 应该为空	测试 iOS 和 Chrome 时值为 Safari，测试 Android 时值为 Browser, null (默认)
newCommandTimeout	否	Appium 服务器等待 Appium 客户端发送新消息的时间，单位为秒，超过设置的时间没有收到新消息时，Appium 服务器会认为客户端退出了	60 (默认)
autoLaunch	否	Appium 服务器是否默认安装被测应用程序并且自动启动该程序	true (默认)
language	否	(仅模拟器使用) 设置模拟器使用的语言	例如: fr, null (默认)
locale	否	(仅模拟器使用) 设置模拟器的国别	例如: fr_CA, null (默认)
udid	否	(仅真机测试时使用) 测试设备的 ID	在多设备同时与一台电脑连接时必须制定
orientation	否	(仅模拟器使用) 设置模拟器启动时的屏幕方向	LANDSCAPE 或 PORTRAIT, null (默认)
autoWebview	否	直接切换到 WebView 的上下文	false (默认), true
noReset	否	在一个 Session 开始前不重置被测程序的状态	false (默认), true
fullReset	否	iOS 通过删除模拟器目录来重置程序状态。Android 通过卸载程序的方式重置程序状态，而且在 session 结束的时候还会卸载程序	false (默认), true

表6-2 仅对Android测试有效的设置

Capability 名称	是否为必填项	描述	值
appActivity	是	被测应用程序启动的 Activity 的名称。该名称在程序启动的时候从 Logcat 日志中找到，详细获取步骤可以参考 6.2.2 节。如上一节中电话本的 Activity 在 Logcat 显示为 “com.android.contacts/.activities.PeopleActivity”，则对应的 appActivity 为 “.activities.PeopleActivity”	com.android.contacts/.activities.PeopleActivity 或者 .activities.PeopleActivity

(续)

Capability 名称	是否为必填项	描述	值
appPackage	是	被测应用程序的包名。同 appActivity 的获取方法一样，但是去 “/” 之前的文本，如电话本启动时日志为 “com.android.contacts/.activities.People-Activity”，那么包名需要写 “com.android.contacts”	例如：com.android.contacts
appWaitActivity	否	测试时需要等待显示的 Activity 的名称	例如：SplashActivity，null（默认）
appWaitPackage	否	测试时需要等待运行的包名称	null（默认）
deviceReadyTimeout	否	等待测试设备 Ready 的超时时间，单位为秒	5（默认）
autoWebViewTimeout	否	等待 WebView 的 Context 变为 active 的时间，单位为毫秒	2000（默认）
unicodeKeyboard	否	是否支持 unicode 的键盘。如果在测试脚本输入时需要输入中文，请将该设置设为 true	false（默认），true
resetKeyboard	否	是否在测试结束后将键盘重置为系统默认的输入法。如果该 resetKeyboard 为 false，那么在测试结束后输入法还是 Appium 的测试输入法，测试手动输入时不会弹出键盘，需要进入系统设置将输入法改回来才能正常输入。但是如果 resetKeyboard 为 true 的话，每个 session 结束后都会重置键盘，测试时间会增加	false（默认），true
dontStopAppOnReset	否	在通过 ADB 启动其他被测程序时，保持当前正在运行的应用程序的进程。另外，如果被测程序是被别的应用程序启动，将 dontStopAppOnReset 设置为 false。则 Appium 通过 ADB 启动被应用程序的过程中，可以使正在运行的程序保持为 Alive 状态。简单地说就是 dontStopAppOnReset 设置为 true，Appium 在运行 adb shell am start 命令时不带 -S 标志，否则运行 adb shell am start 命令时会加上 -S 标志	false（默认），true
ignoreUnimportantViews	否	当设置了该 Capability 时，Appium 将调用 UIAutomator 的 ignoreUnimportantViews() 方法，因此 Accessibility commands 在运行时会忽略一些控件，这样可以使测试的运行速度加快。但是这些被忽略的控件将不再对测试可见，这意味着可能导致脚本因为某些控件找不到而失败	false（默认），true
disableAndroidWatchers	否	停止 Android 的 Watcher，Android 将不再监控程序的 ANR 和 Crash，这将减少 Android 的 CPU 消耗。该 Capability 只对基于 UIAutomator 的测试有效	false（默认），true
avd	否	（仅模拟器使用）模拟器的名称	null（默认）
avdLaunchTimeout	否	（仅模拟器使用）等待模拟器启动并与 ADB 连接的时间，单位是毫秒	12000（默认）
avdReadyTimeout	否	（仅模拟器使用）等待模拟器启动动画显示的时间，单位是毫秒	12000（默认）

(续)

Capability 名称	是否为必填项	描述	值
avdArgs	否	(仅模拟器使用) 模拟器启动使用的其他参数	null (默认)
intentAction	否	启动 Activity 的 Intent Action	android.intent.action.MAIN (默认)。更多关于 intent action 的详情参考 Android 源文件的 android.content.Intent 类
intentCategory	否	启动的 Activity 的 Intent 类别	android.intent.category.LAUNCHER (默认)。更多关于 intent category 的详情参考 Android 源文件的 android.content.Intent 类
intentFlags	否	启动 Activity 的 flag	0x10200000 (默认)。更多关于 intent flag 的详情参考 Android 源文件的 android.content.Intent 类
optionalIntentArguments	否	启动 Activity 时其他的一些参数	更多详情参考 Android 源文件的 android.content.Intent 类

以上是现有Capability中的大部分内容，主要是关于Android测试的，还有一部分与iOS相关的Capability就不在这里赘述了，感兴趣的读者请去Appium的官网查看。



提示 笔者主要使用的Capability包括platformName、platformVersion、appPackage、appActivity、unicodeKeyboard、resetKeyboard和newCommandTimeout。这些Capability基本上已经满足了目前的测试需求。

6.2.4 Appium API的解读

Appium的客户端（WebDriver）提供的接口按作用大致可以分为控件的查找、手势操作和系统操作，本小节将对测试过程中最常用的部分方法进行阐述，如果使用的方法在本文中没有提到，请查阅WebDriver的帮助文档或者源代码获取详细信息。

1.控件查找API

WebDriver提供的方法可以根据ID、Xpath、Name、Class Name、Accessibility id和UIAutomator来查找控件，详细的方法和参数说明见表6-3。

表6-3 查找控件的方法和参数说明

API	方法描述
find_element_by_id(self, id_) find_elements_by_id(self, id_)	通过控件的 resource id 来查找控件，resource id 可以通过 uiautomatorviewer 或者 Appium 的 Inspector 查看
find_element_by_xpath(self, xpath) find_elements_by_xpath(self, xpath)	根据 XPath 来查找控件，Android Native App 一般很少使用该方法，但常用于 Web App 和 Hybrid App 测试中
find_element_by_name(self, name) find_elements_by_name(self, name)	在 Native App 测试中，Name 参数就是控件的 Text
find_element_by_class_name(self, name) find_elements_by_class_name(self, name)	在 Native App 测试中，参数 Name 指代控件的类型，如 android.view.Text；在网页测试时指代网页 element 的属性类名，如 <div class="highlight-java" style="display: none; ">...</div> 中，class name 为 “highlight-java”
find_element_by_android_uiautomator(self, uia_string) find_elements_by_android_uiautomator(self, uia_string)	根据 UIAutomator 的语法查找控件，该方法是 Web-Driver 在兼容 Appium 时才新加的方法
find_element_by_accessibility_id(self, id) find_elements_by_accessibility_id(self, id)	根据控件的 accessibility ID 来查找，accessibility ID 指 Native App 控件的 Content Description；若列表的项 ID 信息可能都一样，则可以让开发人员为每个列表添加一个 Content Description 用于列表项的查找
find_element_by_link_text(self, link_text) find_elements_by_link_text(self, link_text)	根据链接的文本查找控件，仅用于 Web App 和 Hybrid App 的测试
find_element_by_partial_link_text(self, link_text) find_elements_by_partial_link_text(self, link_text)	根据链接的部分文本查找控件，仅用于 Web App 和 Hybrid App 的测试
find_element_by_css_selector(self, css_selector) find_elements_by_css_selector(self, css_selector)	根据网页 element 的 CSS Selector 查找控件，仅用于 Web App 和 Hybrid App 的测试
find_element_by_tag_name(self, name) find_elements_by_tag_name(self, name)	根据网页 element 的 Tag 查找控件，仅用于 Web App 和 Hybrid App 的测试



注意 页面中同一个ID的控件可能不止一个，最常见的情况就是列表项，笔者项目中同一个列表的列表项的ID都是一样的。表6-3中的find_element_by_id是查找页面中第一个ID为指定参数的控件，返回一个控件；而find_elements_by_id是查找页面中所有ID为指定参数的控件，返回包含所有满足条件的控件列表。其他的方法也是同样的原理。

2.获取和操作控件信息的API

测试过程中常常需要获取控件的信息来进行测试验证。WebDriver 有一个类叫作WebElement，所有的控件都是该类的对象，它提供一些专门的API用于获取控件信息，见表6-4。

表6-4 获取控件信息的API

API	方法描述
text(self)	获取控件显示的文本信息，如 <code>element.text</code>
tag_name(self)	获取控件的 Tag 名称，主要是在网页测试时使用，如 <code><input></input></code> 返回的 Tag 名称为 <code>input</code>
click(self)	点击控件，如 <code>element.click()</code>
clear(self)	如果是一个文本输入控件的话，该方法可以清除控件的文本，如 <code>element.clear()</code>
is_enabled(self)	判断控件是否可用了，如果是可用的，则返回 <code>true</code>
is_selected(self)	判断控件是否被选中了，如果被选中了则返回 <code>true</code>
is_displayed(self)	判断控件是否显示，如显示则返回 <code>true</code>
get_attribute(self, name)	获取控件某项属性的值，如果该属性不存在，则会返回 <code>None</code> ，如 <code>element.get_attribute("enabled")</code> 等同于 <code>is_enabled()</code> 方法
parent(self)	返回控件的父控件，返回值为一个控件对象
send_keys(self, *value)	模拟输入文本到控件中，Value 为输入的文本串信息，如 <code>textElement.send_keys("腾讯地图")</code>

3.手势操作API

WebDriver为了支持Appium手机自动化测试，新增加了手机上的手势操作方法。这些方法包括点击、滑动屏幕、放大缩小、拖曳以及滚动屏幕等。表6-5描述了手势操作方法的功能和参数。

表6-5 手势操作方法的功能和参数

API	方法描述
<code>tap(self, positions, duration=None)</code>	点击屏幕上的位置，最多可以 5 个手指同时点击； <code>positions</code> 是一个列表，每一个列表项是一个二元组，值分别是屏幕上坐标的 X 和 Y； <code>duration</code> 为点击的时间长短，单位为毫秒。如果该参数不提供，则认为是点击操作，如果该参数给定参数，则被 <code>WebDriver</code> 识别为长按操作，如 <code>driver.tap([(100, 20), (100, 60), (100, 100)], 500)</code>
<code>swipe(self, start_x, start_y, end_x, end_y, duration=None)</code>	从屏幕上 A 点滑动到 B 点， <code>duration</code> 为滑动动作执行的时间，单位为毫秒
<code>flick(self, start_x, start_y, end_x, end_y)</code>	从屏幕 A 点快速地滑动到 B 点
<code>pinch(self, element=None, percent=200, steps=50)</code>	在某控件上执行缩小操作，默认缩放比例为 200%， <code>step</code> 表示缩小动作分多少步完成，默认为 50
<code>zoom(self, element=None, percent=200, steps=50)</code>	在某控件上执行放大操作，默认缩放比例为 200%， <code>step</code> 表示放大动作分多少步完成，默认为 50
<code>scroll(self, origin_el, destination_el)</code>	从 <code>origin_el</code> 控件滚动到 <code>destination_el</code> 控件，参数必须是两个控件，而不是控件信息
<code>drag_and_drop(self, origin_el, destination_el)</code>	把 <code>origin_el</code> 控件拖曳到 <code>destination_el</code> 的位置

4.系统操作API

`WebDriver`提供的系统操作API是用于模拟操作硬件、设置系统环境或者获取系统信息的方法，如按返回键、设置网络 and 文件操作等。表6-6包含了现有系统操作方法的描述和参数说明。

表6-6 现有系统操作方法的描述和参数说明

API	方法描述
<code>contexts(self)</code>	获取当前会话 Session 所有可用的上下文 (Context)。关于 Context 的使用将在 6.3.3 节中详细阐述
<code>current_context(self)</code> <code>context(self)</code>	获取当前会话 Session 正在使用的上下文
<code>keyevent(self, keycode, metastate=None)</code>	模拟发送一个硬键码到手机, Selendroid 使用该方法可以模拟硬件操作, 如模拟按下返回键 <code>driver.keyevent(4)</code> 。详细的 keycode 请通过网址 http://developer.android.com/reference/android/view/KeyEvent.html 查询
<code>press_keycode(self, keycode, metastate=None)</code>	模拟发送一个硬键码到手机, 如模拟按下返回键 <code>driver.press_keycode(4)</code>
<code>long_press_keycode(self, keycode, metastate=None)</code>	模拟发送一个长按硬键码到手机
<code>current_activity(self)</code>	获取当前正在显示的 Activity 信息
<code>reset(self)</code>	重置当前被测程序到初始状态
<code>pull_file(self, path)</code>	拉取手机上的一个文件, 并以 Base64 格式编码返回文件数据; <code>path</code> 为手机上的文件路径
<code>pull_folder(self, path)</code>	拉取手机上的一个目录, 目录的所有内容会被压缩打包, 并以 Base64 格式编码返回数据; <code>path</code> 为手机上的目录路径
<code>push_file(self, path, base64data)</code>	将一个 base64 格式编码的数据推送到手机的文件路径; <code>path</code> 为手机上的文件路径, <code>base64data</code> 为要推送的数据
<code>background_app(self, seconds)</code>	将被测应用程序放到后台持续运行一段时间, <code>seconds</code> 为持续的秒数
<code>is_app_installed(self, bundle_id)</code>	判断应用程序是否安装。 <code>bundle_id</code> 是被查询的应用程序的 ID, 在 Android 设备上, <code>bundle_id</code> 是应用程序的完整包名, 如 “com.android.contacts”
<code>install_app(self, app_path)</code>	将 <code>app_path</code> 路径的应用程序安装到手机上。此处的 <code>app_path</code> 需要包含目录和文件名, 如 “C:/test.apk”
<code>launch_app(self)</code>	在测试设备上启动 Desired Capabilities 中指定的应用程序
<code>close_app(self)</code>	如 Desired Capabilities 指定的应用程序正在运行, 则关闭该程序
<code>start_activity(self, app_package, app_activity, **opts)</code>	启动某个 Activity, 如果该 Activity 不属于另外一个未启动的程序, 那么该程序将被启动, 然后该 Activity 被打开。该方法的使用可以参考 <code>adb shell am start</code> 命令
<code>shake(self)</code>	模拟晃动手机的事件

(续)

API	方法描述
open_notifications(self)	打开消息通知栏，该方法仅对 Android API Level18 以上的系统有效
network_connection(self)	返回当前网络连接的类型，网络类型参考 appium.webdriver.ConnectionType
set_network_connection(self, connectionType)	设置网络， connectionType 的值为： Value (Alias) Data Wifi Airplane Mode ----- 0 (None) 0 0 0 1 (Airplane Mode) 0 0 1 2 (Wifi only) 0 1 0 4 (Data only) 1 0 0 6 (All network on) 1 1 0
get_screenshot_as_file(self, filename)	将手机屏幕截图并保存为电脑上的文件， filename 为文件的路径和名称

6.3 Appium框架在腾讯地图中的实践

6.3.1 Appium接口的封装

Appium的自动化测试的过程大部分都是由建立Appium的连接、查找控件、操作控件、等待控件的显示、获取控件信息和测试验证等步骤组成的。在地图项目中，笔者对常用的WebDriver接口进行了封装，将这些方法放到了一个UiHelper的类中进行统一管理，便于简化测试脚本，降低开发和维护的成本。

1.Appium连接建立与断开方法的封装

在6.2.2节的示例脚本中，脚本在建立连接前都提供一个Desired Capabilities的字典数据，并用它初始化了与Appium的连接。如果读者有更多的测试用例，大家应该可以发现初始化Appium连接的代码的每个用例都是必需的，但是Desired Capabilities基本上很少改变，而且这部分代码都是重复的。因此笔者想到将Desired Capabilities的数据独立存放到文件中进行管理，这样既可以重用，也便于维护。Desired Capabilities数据的文件如图6-10所示。

```
platformName=Android
platformVersion=4.4
deviceName=052abd630a670a6a
#此处的app不需要，因为需要动态的安装最新的app|
#app=
appPackage=com.android.contacts
appActivity=.activities.PeopleActivity
unicodeKeyboard=true
newCommandTimeout=150
remoteHost=http://127.0.0.1:4723/wd/hub
```

图6-10 Desired Capabilities配置文件示例

既然Desired Capabilities存放到了独立的文件内，那么Appium的初始化代码也需要做相应的变更。笔者将Desired Capabilities数据的读取放到了UiHelper这个类的__init__方法中，对Appium连接的建立和断开也进行了封装，用例可以直接使用UiHelper的方法来完成Appium连接的建立，如代码清单6-7所示。

代码清单6-7 使用UiHelper建立与Appium服务器的连接示例

```
def __init__(self, configPath):
    self.desired_caps = {}
    self._driver = None
    file_object = open(configPath)
    try:
        for line in file_object:
            #'#'的行为注释
            , 忽略

            if(line.startswith("#")):
                continue
            line = line.strip()
            words = line.split("=")
            if(len(words) != 2):
                continue
            if(words[0] == 'app'):
                self.desired_caps[words[0]] = PATH(words[1])
```

```
        elif(words[0] == 'remoteHost'):
            remoteHost = words[1]
        else:
            self.desired_caps[words[0]] = words[1]
    finally:
        file_object.close( )
def initDriver(self):
    self._driver = webdriver.Remote(self.remoteHost, self.desired_caps)
def quitDriver(self):
    if(self._driver):
        self._driver.quit()
```

2.控件查找方法的封装

在6.2.4节中详细描述了控件查找的方法，在笔者项目中的测试脚本主要通过XPath、ID、Name和ClassName来查找控件。WebDriver提供的查找方法各不相同，但是对于测试脚本的开发人员来说，使用这些方法都是为了查找控件，因此笔者考虑将常用的查找方法融合到一起，脚本开发人员不用显示调用WebDriver中具体的方法来查找控件，只需要调用同一个方法就足够了。

经过对WebDriver的接口进行分析，发现这些接口需要传入一个字符串，只是传入字符串的格式有一些不同。例如，通过ID来查找，参数字符串中会包含“: id/”这样的字符串；而通过XPath来查找，参数则以“//”开始；对于通过Name或者ClassName的方法，参数没有明显的特征，但在实际测试过程中发现使用Name来进行查找的次数明显比较多。所以当判断出不是XPath或者ID时，直接通过Name查找一次，如果还找不到控件，再尝试通过ClassName来查找。封装后的方法如代码清单6-8所示，该方法用于在当前页面中查找单个控件。

代码清单6-8 自己封装的控件查找方法

```
def findElement(self, controlInfo):
    element = ""
    if(controlInfo.startswith("//")):
        element = self._driver.find_element_by_xpath(controlInfo)
    elif(":id/" in controlInfo or ":string/" in controlInfo):
        element = self._driver.find_element_by_id(controlInfo)
    else:
        #剩下的字符串没有特点，无法区分，因此先尝试通过名称查找

        try:
            element = self._driver.find_element_by_name(controlInfo)
        except:
            self.logger.logDebug("Cannot find the element by id, try class name")
            #如果通过名称不能找到，则通过

class name查找

        element = self._driver.find_element_by_class_name(controlInfo)
        #self.logger.logDebug("Found the element by: " + controlInfo)
        return element
```

在测试脚本中大部分控件查找都只需要调用该方法即可实现，此外在当前页面中查找一组控件的方式也是同样的原理。调用本方法的示例如代码清单6-9所示。

代码清单6-9 使用自定义的方法进行控件查找

```
#初始化

UiHelper对象，并使其与

Appium建立连接

uiHelper = UiHelper("deviceConfig.txt")
uiHelper.initDriver()
#通过

ID来查询控件

createContactButton = uiHelper.findElement(
    "com.android.contacts:id/menu_add_contact")
```

```
#通过控件文本来查找控件

name = uiHelper.findElement(u"姓名
")
#通过控件的类名 (
ClassName) 来查找画面中第一个该类型的控件

editText = uiHelper.findElement("android.widget.EditText")
#通过
Xpath查找控件，类似于控件位置的绝对路径
, 较难维护

#该方法一般在
Android上很少使用

view = uiHelper.findElement(
"//android.widget.FrameLayout[0]/android.view.View[0]")
```

以上方法是在当前页面查找控件，在测试中还有另外一些需要在已知的某个控件中查找控件的情况。如图6-11所示，当判断北京市的离线地图是否下载时，需判断列表项中右边的状态显示为“已下载”。然而“北京市”字样的控件与右边的“已下载”字样的按钮不在一个TextView中，但是这两部分控件有一个共同的父控件，所以测试脚本首先要查找某个列表项中包含“北京市”，然后再在列表项中判断“已下载”就能找到，这个流程就需要使用到在某控件中查找子控件的方法。笔者将这类特殊作用的方法也封装在一起，如代码清单6-10所示。

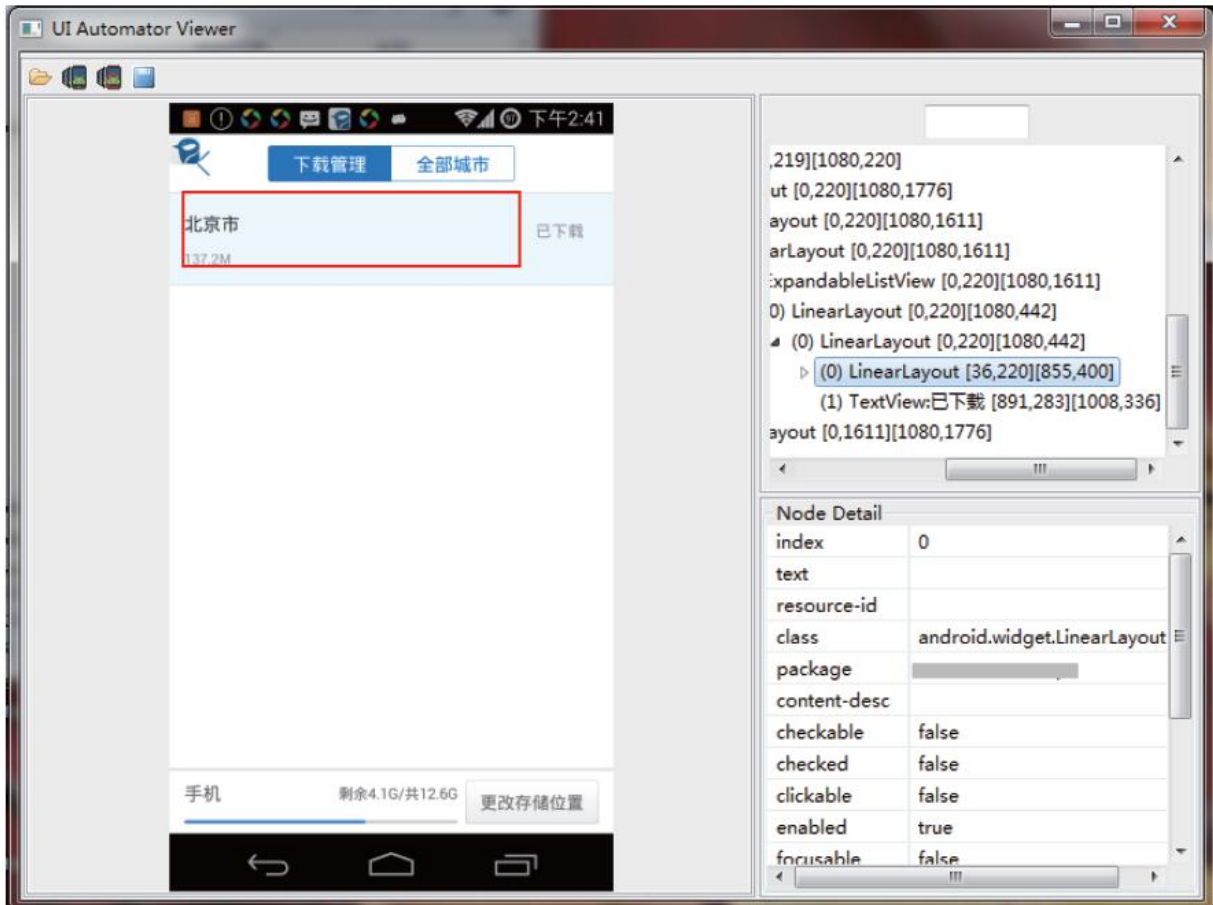


图6-11 离线地图下载控件布局示意图

代码清单6-10 在控件中查找其他控件的方法封装

```
def findElementInParentElement(self, parentElement, childElementInfo):
    element = ""
    if(childElementInfo.startswith("//")):
        element = parentElement.find_element_by_xpath(childElementInfo)
    elif(":id/" in childElementInfo):
        element = parentElement.find_element_by_id(childElementInfo)
    else:
        #此处省略其他方式的代码

    return element
#调用示例如下:

Parent = uiHelper.findElement("com.XXXX.XX:id/listItem")
statusText = uiHelper.findElementInParentElement(
    Parent, "com.XXXX.XX:id/status")
```

3.UI等待方法的封装

在6.2.2节结束时提到HelloWorld的脚本中的不足之处就是等待的时间是固定的，在这里我们将解决这个问题。在UI自动化测试中，等待某状态到来是很常见的情况，比如在UI上操作了某控件后等待某个Activity或控件的显示，然后进行下一步操作。WebDriver提供了`implicitly_wait`方法可以在一定时间内等待控件的显示，此外Python语言提供的`sleep`方法也可以用于等待，这些方法都是等待固定的时间。脚本运行究竟需要等待多长时间明显是不能确定的，例如在网络不稳定时等待请求返回的时间会更长，另外不同的测试设备性能不一样，等待的时间也各不相同。如果设置一个足够长的时间当然可以减少测试的失败，但是我们往往希望测试脚本能尽可能快地运行，每次运行都固定等待很长时间就不明智了。

WebDriver提供了一个叫作WebDriverWait的等待方法类。该类可以提供Until方法和until_not方法，这两个方法需要用户指定等待条件和等待时间。Until方法在条件得到满足时就会中断等待，继续后续的步骤；而until_not方法是在条件不满足时中断等待。代码清单6-11所示使用WebDriverWait的方法等待`com.android.contacts: id/left_button`控件的显示，最长等待10秒。

代码清单6-11 WebDriverWait方法的使用示例

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
#do something then wait for an element shown.....

WebDriverWait(self._driver, 10).until(EC.visibility_of_element_located((By.ID,
"com.android.contacts:id/left_button")))
```


如上面的代码所示，**WebDriverWait**的使用需要用到 **expected_conditions.py** 中的各种查询条件，包括判断控件是否显示、文本是否正确或者各种状态是否与期望状态一致等，这些查询条件同样可以用于 **Assert** 方法中。详细的查询条件这里就不再赘述了，感兴趣的读者可以通过查看 **WebDriver** 的文档或者 **erexpected_conditions.py** 源码了解。

在地图项目中，笔者封装了一些简单的等待方法，原理与 **WebDriverWait** 的原理类似，都是将固定等待的过程分成了多次 UI 的检查过程，封装后的方法如代码清单 6-12 所示，该方法参数需要提供等待的控件信息和等待的秒数，每隔一秒就会检查一下期望的控件能否被查询到，如果查询到了就立即返回；如果没有查询到则继续等待，直到超时。对于等待某个控件消失也可以采用同样的方式。

代码清单 6-12 等待方法的封装

```
def waitForElement(self, elementInfo, period):
    for i in range (0, period):
        sleep(1)
        try:
            self.findElement(elementInfo)
            return
        except:
            continue
    raise Exception("Cannot find %s in %d seconds" %(elementInfo,period))
```

以上是针对控件显示的等待方法，简单且实用。另外，在一些特殊情况下，比如在当前画面有一个控件ID叫作“okButton”，单击该按钮时会跳转到另一个画面，在新的画面中又有一个控件ID叫“okButton”。此时，如果通过该控件是否显示再判断是否该进行下一步操作，脚本就可能出错。在这种情况下，可以使用WebDriver提供的wait_activity的方法来等待画面显示。

 **提示** 使用wait_activity方法时需要知道等待的画面是什么Activity，笔者一般会手动点到需要等待的画面，然后通过以下的adb命令来获取当前Activity：`adb shell dumpsys activity|findstr“mFocusedActivity”`。

对比WebDriverWait和笔者实现的方法，原理都一样，效率也相差不多，但是笔者没有实现对状态的检查等，因此，WebDriverWait更加全面一些。建议读者使用WebDriverWait，但最好将WebDriverWait方法进行二次封装，让其使用起来更加简单一些。

4.控件信息验证方法的封装

在测试脚本中少不了对控件信息或者状态的获取，这些方法大量使用在脚本的验证过程中。如上一部分讲等待时提到的expected_conditions就比较好用。笔者也将这部分高频率使用的方法进

行了简单封装。获取控件的状态的步骤比较简单，首先找到对应的控件，然后查找控件的状态。笔者封装的方法包括 `checkElementIsShown`、`checkElementShownInParentElement`、`checkElementIsSelected`、`checkElementIsChecked`、`checkElementIsEnabled`等，这些方法返回值为True或False，在if语句或者Assert中使用起来比较方便，以下代码是`checkElementIsEnabled`的示例。

```
def checkElementIsEnabled(self, elementInfo):  
    element = self.findElement(elementInfo)  
    return element.get_attribute("enabled")
```

以上为部分常用的Appium接口的封装方法，其他一些方法就不在这里赘述了，详情可以参考TMQ官网下载的第6章中的UIHelper的源代码。

6.3.2 测试脚本设计思想

在开始讲解本部分内容前，我们先思考一个问题。如果一个月发布一个版本，在上线前都需要回归某功能，如果实现这个功能的自动化只需要一天，那是否应该对这个功能实现自动化测试？这个问题没有绝对的答案，与实际项目的具体情况有很大的关系。其实笔者希望读者在看到这里时，心里对自动化测试有一个正确的概念：“自动化测试的根本目的是提高效率和降低成本。”在实施自动化测试之前，我们需要进行如下思考：

首先，项目是否真的需要自动化测试，投入产出比如何？

其次，什么自动化方法更适合？

最后，如何实现自动化？

在实际项目中，很多测试人员过多地考虑第三个问题，也会做一些关于第二个问题的调查，但往往缺乏对第一个问题的思考。在此建议读者从自己项目的角度出发，慎重思考自动化测试的投入和收益，选择适合项目的方法，使投入产出比最大化。

在这里，我们不过多地讨论是否需要自动化。在已经确定自动化测试可以带来收益的情况下，则需要选择测试方案，尽量让开发成本

和维护成本降低一些。如果考虑选择Appium作为自动化测试工具，建议读者考虑以下几个方面：

第一，被测试程序主要变化的地方是什么，是否适合用UI自动化测试。如果应用程序UI变化概率比较小，代码变动主要是下层逻辑，这样的程序比较适合做UI自动化测试。如果UI变化大，那么UI自动化脚本维护成本就会很大，自动化测试投入产出比不高。

第二，被测试的程序是什么类型的应用。比如游戏类的测试，可能很多的画面都是通过OpenGL直接渲染的，Appium无法找到OpenGL直接渲染出来的画面里的元素，而且从UI上去验证游戏画面非常困难，在这种情况下，如果通过UI实施自动化测试可能需要大量的后期人工检查。

第三，自动化测试的目标是什么，是否对测试的运行时间有要求。如果自动化的目标是快速地回归，要求测试脚本短时间内完成大批脚本的运行的话，此时可能不适合用Appium。因为Appium是UI自动化测试，UI自动化测试的运行同一条测试的时间比人工执行的时间要长，所以很难在短时间内运行大批量的测试。但如果没有时间要求，比如每天晚上定时运行的冒烟测试，则不用考虑时间效率。

第四，自动化测试是否要脱机执行。比如性能测试中的耗电量测试，必须断开与电脑的连接，否则USB线会给手机充电。由于Appium

是必须与电脑连接的，以上的场景就不能通过Appium来实施自动化，可以考虑选择UIAutomator。

第五，如果选择Appium来实施自动化测试，什么语言比较合适。可以从当前团队成员的能力考虑，选择学习成本和实施成本较低的语言。

测试方案确定下来后，就需要考虑如何实施了。有过自动化测试开发经验的读者应该知道，自动化测试的脚本开发其实不难，但测试脚本的维护却是比较困难的。测试脚本设计的思想是尽量地提高测试脚本的可重用性和稳定性，降低脚本的维护成本，提高收益。

6.3.3 Appium在腾讯地图中的测试实践

腾讯地图项目在初期的时候，由于功能较少，测试人员较容易发现开发**check in**代码引入的回归问题。但是随着项目功能的增多，测试人员应对新需求测试压力越来越大，对老功能的回归测试的频率从最开始的一周两次，变成每个月只在发版前才进行几轮，项目组也开始挑战测试团队发现Bug晚的问题。基于这样的背景，测试人员开始思考是否可以通过自动化方式来尽早发现这些问题，特别是老功能的回归测试。

腾讯公司有一个叫作**RDM**的集成系统，该系统可以定期对最新的代码进行编译，项目成员可以获取每天最新的版本（**Daily Build**）。测试团队经过与开发人员的沟通，最终决定采用基于**RDM**系统的**Daily Build**实施冒烟测试。

有了这样的想法之后，我们就开始调研测试方案。首先我们调研了是通过接口去实施自动化，还是从**UI**上去实施自动化。在调研过程中，我们发现通过接口的方式去实施自动化的优缺点，优点是测试运行比较快，可以很快地进行回归；缺点是测试代码对开发代码有较大的依赖，往往是开发的代码发生了改变，测试代码没能同步修改，等下一次编译运行时发现测试代码又编译不通过。测试用例少的时候还好，随着用例的扩充，测试人员已经疲于维护测试代码了。之后我们

分析了地图项目组的情况，发现从地图4.0版本以来，UI几乎没有大的改动，多数的改动都是发生在逻辑层的，因此最终我们决定从UI上实施自动化。当然从UI上实施自动化测试也存在一定的问题，比如运行效率较低，此外地图一般都是通过OpenGL直接绘制出来的图片，无法自动化验证，等等。对于效率低的问题，我们采用晚上运行的方案，即让RDM系统在晚上编译，冒烟测试在晚上运行，第二天再看测试结果，这样即使运行效率低点儿，也不占用人的时间，还是可以接受的；对于验证的问题我们最终采用折中的方案，就是自动化测试能验证多少就验证多少，如果需要在地图上进行验证，那就让外包人员人工验证，一个熟练的外包人员看结果只需要10分钟就能完成了。

当主思路确定后就开始测试方法的选择，我们考察了UIAutomator、Robotium和Appium等工具，这三个工具基本上都能满足地图项目的需求，但最终项目组选择了使用Appium。选择Appium除了上面的思考以外，还有以下一些考虑：

- 自动化工具简单易用，最好是学习成本不太高，便于调试。以上三个工具基本上都符合，但笔者觉得还是Appium最简单，大部人员都能很快学会。

- 该测试不仅要用于对Daily Build的测试，还要用于发布前的测试，测试方案最好不需要对测试包进行修改。其中Robotium需要对测试包做一点儿修改，在这里不太符合。

·由于地图组分为Android和iOS的冒烟测试都是一片空白的，需要从头建立，最好有一种方案可以让两个系统共用一部分脚本，降低开发成本，且统一输出格式，以便于统一展示。关于这一点，只有Appium满足条件。

Appium支持多种开发语言，但是实施自动化时，我们选择Python作为开发语言，主要是项目组会Python的人挺多，学习成本低；而且Python是解析执行的语言，修改后不需要编译立刻可以运行，调试也较为方便。

由于UI自动化的执行效率实在不快，我们并不期望冒烟测试可以替代大部分的回归测试工作，而且考虑到脚本覆盖过细，场景后期维护成本会很高，因此我们的冒烟测试用例只覆盖了各功能主要的路径。

基于Appium的框架，地图测试用Python实现了一个通用的测试模块，可以在Android和iOS上共用。该测试模块的类图如图6-12所示。

测试脚本的基类TestBase继承于Python的unittest.TestCase，在TestBase中将初始化UiHelper，以及测试日志记录的类Logger。

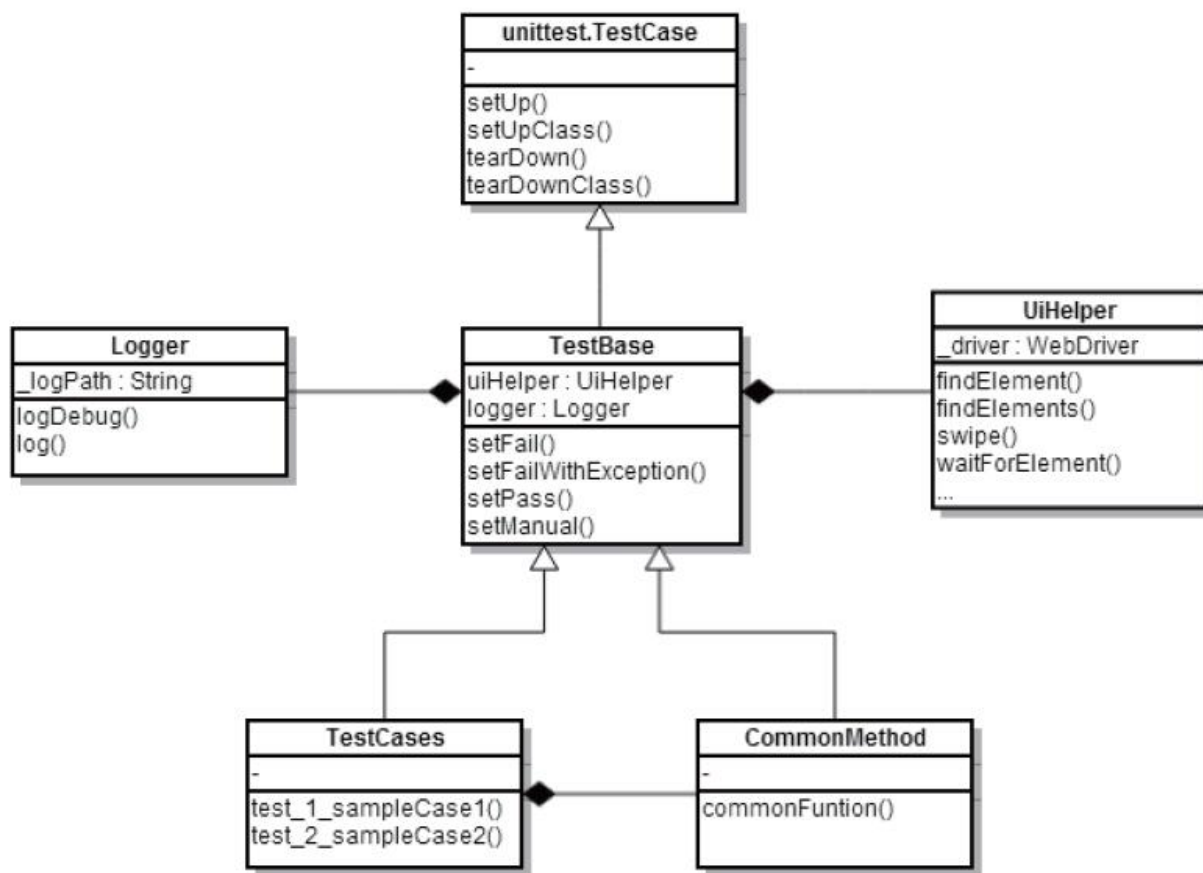


图6-12 地图冒烟测试模块的类图

`UiHelper`是唯一与Appium服务器进行通信的类，所有与Appium服务器相关的操作都统一使用该类。

`Logger`是一个单例的对象，用于记录格式化的测试日志，所有测试运行中需要输出的日志都将用这个类的方法来记录。

测试脚本分为两部分，它们都继承于`TestBase`。一部分是测试用例（`TestCases`），它是具体的测试场景；另外一部分是在多个用例中都使用到的公共方法库（`CommonMethod`）。

1.可重用性

提高测试脚本的可重用性可以一定程度上降低开发和维护成本。在地图项目中，脚本可重用性体现在测试代码重用和测试脚本模板重用两方面。

测试代码的可重用主要是公共过程的提取。有自动化测试经验的读者应该知道，测试工具一般会提供`setUp`和`tearDown`两个方法，`setUp`方法会在用例方法运行前被执行，`tearDown`方法会在用例方法运行后被执行。Python的`unit.TestCase`类也提供同样的方法。因此将Appium连接的建立和网络初始化放到这个方法中，就不用再在每个脚本中重复开发，如代码清单6-13所示。在地图项目中，测试场景中常用的方法也被提取出来，形成公共方法库，测试脚本可以由多个公共方法组合形成用例。

代码清单6-13 基类中公共方法示例

```
def setUp(self):
    try:
        self.uiHelper.initDriver()
        self.uiHelper.enableWifiOnly()
    except:
        self.uiHelper.quitDriver()
        exstr = traceback.format_exc()
        self.logger.log(exstr)
        self.logger.log("Setup Failed")
        self.fail("")
def tearDown(self):
    try:
        self.uiHelper.quitDriver()
        sleep(5)
    except:
        pass
```

测试脚本模板重用是指抽取脚本通用部分，开成公共模板，便于后期快速开发脚本，如代码清单6-14所示，并且这些模板格式较为固定，能通过脚本直接生成。这样测试人员就可以重点关注用例的开发，缩短测试脚本的开发时间。

代码清单6-14 测试脚本模板的示例

```
def test_1_sample(self):
    caseName = self.get_current_function_name()
    decription = "the description of this case"
    try:
        self.markCaseStart(self.currentFileName, self.__class__.__name__,
        caseName, decription)
        CommonMethod.dismissUserGuideIfExist()
        CommonMethod.dismissDialogIfExist()
        #Auto generated code
        self.saveScreenshot(caseName + "_1.png")
        self.setPass()
    except Exception as e:
        self.setFailWithExceptionInfo(caseName)
```

2.稳定性

Appium的自动化测试在运行过程中遇到的稳定性问题主要有以下几种情况：

(1) 用例运行时受前一个用例运行的影响，程序状态不正确导致测试脚本运行错误。最常遇到的是前一个脚本异常退出，**Session**还没有正常结束，可能导致后面用例在建立**Session**时失败。

(2) 脚本包含多个用例，如果用例查找控件失败，则会抛出异常，导致整个脚本都退出了，后续用例将不能正常运行。例如，在刚

开始的时候脚本没有做任何异常捕获，当用例出现异常时直接就中断了该脚本的运行，后面的用例就不可能再被运行了。

(3) 在程序运行过程中，未预期的消息框弹出，导致测试脚本运行出错。例如我们比较容易遇到启动地图时提示用户需要开启Wi-Fi或者GPS的情况。

针对第一种情况，笔者的解决方法如下：

首先，用例之间不相互耦合，每一个用例都是一个可独立运行的方法，不依赖任何其他用例。

其次，每个测试脚本都从程序启动状态开始运行，因为中间状态不能确保正确。如果用例会对程序有一些设置，那么每个用例结束后将设置状态恢复到初始状态。

最后，在setUp和tearDown方法中初始化与断开Appium的连接，保证Appium服务器是一个清洁的环境。

针对第二种情况，笔者的解决方法是将每一个测试脚本的代码都包含在try-except中，保证测试脚本中的异常都能被每个用例捕获，即使当前用例失败了，后续用例也可以正常运行。

对于第三种情况笔者也没有很好的方法，只能尽量保证测试环境的清洁，在setUp方法中就将Wi-Fi和GPS打开，并在启动地图后稍作等

待，然后判断是否存在Dialog，有则清除掉，降低这类问题出现的概率。

3.可维护性

UI自动化测试脚本的维护主要发生在UI变更时。UI的变更主要有两个方面：一是功能的流程发生变化，二是控件的信息（如ID）发生变化。


测试过程提取为公共方法的作用之一就是降低过程变化时的维护成本。当流程发生变化时，只需要改动公共方法，而不用对每个脚本进行维护。最常见的情况就是每个用例都需要处理新功能引导页，而且每个版本发布时新功能引导页都可能会变化。如果将该方法提取出来，则当新功能引导变化时，只需要维护该方法即可。

而针对控件信息发生变化的情况，笔者采用的方法是将控件的信息集中维护到一个常量文件中（图6-13），当控件信息发生变化时，只需要维护该文件即可，不用修改用例脚本。



图6-13 控件信息通过文件独立管理示例

另外，关于用例的管理，笔者主要是通过物理文件来进行的，比如冒烟测试的用例会存放在一个目录下，而功能测试的脚本又存放在另外的目录中。而目录中的测试脚本又按功能模块分为不同的脚本文件，在每个脚本文件中有一个类，该类包含多个测试用例。而我们的脚本运行工具是Nosetests，该工具可以通过命令的参数来指定运行的范围。

 **提示** Python安装成功后，Nosetests可以通过以下方式安装：

命令行下运行： `pip install nose` 。

Nosetests的运行方式有：

运行指定路径下所有用例，如`nosetests-s D: \map\bvt\`;

运行指定文件内的所有用例，如`nosetests-s D:`

`\map\bvt\test_map_bvt.py`;

运行指定类下所有用例，如`nosetests-s D:`

`\map\bvt\test_map_bvt.py: MapTest_BVT`;

运行指定的用例，如`nosetests-s D: \map\bvt\test_map_bvt.py:`

`MapTest_BVT.test_02_openTraffic_BVT`。

除了运行工具以外，地图测试组还有一个叫作八爪鱼的自动化管理平台辅助进行用例管理。该平台用例管理页面如图6-14所示，在该页面我们可以任意选择需要运行的用例，使用例管理更加方便。

在腾讯地图项目中，自动化脚本都通过SVN部署到一台自动化服务器上，通过八爪鱼平台的任务管理定时将任务发布到自动化机器上去运行，并将自动化结果上传到八爪鱼自动化平台，展示结果如图6-15所示。

在腾讯地图的冒烟测试实施之后，基本上较为严重的回归问题都可以在出现一天之内被发现，甚至后来开发人员已经不满足于一天的延迟时间，因此我们又将冒烟测试任务时间缩短为每4个小时运行一次，更缩短了回归问题的发现周期。

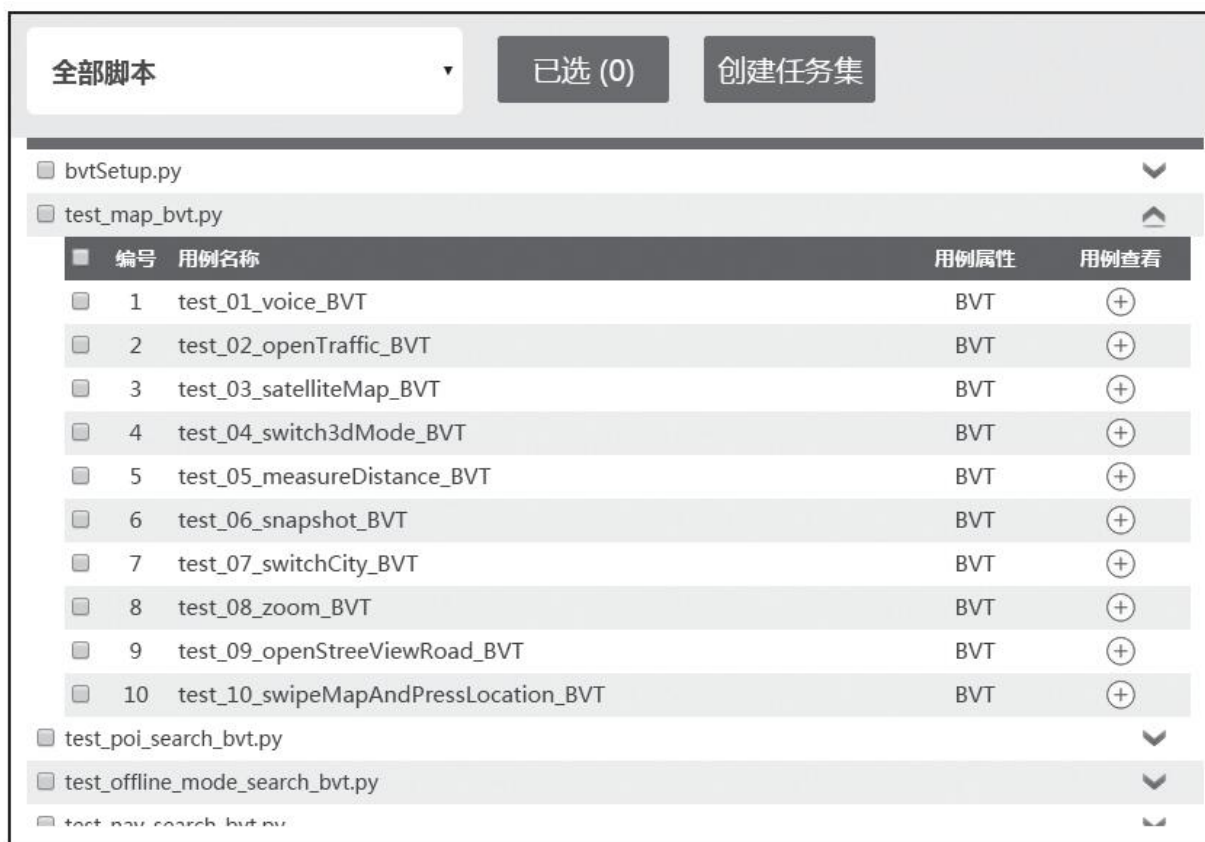


图6-14 八爪鱼平台用例管理页面



图6-15 腾讯地图冒烟测试展示结果

6.3.4 Hybrid App的测试方法

Hybrid App是移动混合应用程序，即在移动应用程序中嵌入了Webview，通过Webview访问网页。移动应用和Webview分别属于两个不同的上下文（Context），移动应用默认的Context为“NATIVE_APP”，而Webview默认的Context为“WEBVIEW_{被测进程名称}”。Appium在测试应用程序时，默认会使用NATIVE_APP的Context；当测试Webview中的网页内容时，需要切换到Webview的Context下；同样如当前正在测试Webview，此时若需要回到移动应用，Context需要切换到NATIVE_APP下。

获取当前手机上正在显示画面的Context可以使用WebDriver提供的contexts方法。调用该方法时如果正在显示的画面包含Webview，从Appium的日志中可以看到可用的Context包含“WEBVIEW”，如下面的日志所示。

```
info: [debug] Getting a list of available webviews
.....
info: [debug] Available contexts:
info: [debug] ["WEBVIEW_com.xxxxx.map"]
info: [debug] Available contexts: NATIVE_APP,WEBVIEW_com.xxxxx.map
```

如果要切换到Webview的Context下，请调用WebDriver中的switch-to.context方法。在UiHelper中调用的代码如下面的代码所示，其中参数

的contextName为上面提到的“Available contexts”中的名称。

```
def switchContext(self, contextName):  
    self._driver.switch_to.context(contextName)
```

在测试Hybrid App时可能会遇到一些常见问题，以下是笔者遇到的问题和这些问题的解决方法。

1.如何获取Webview中的页面信息

当需要操作和验证Webview中的控件时，需要找到这些控件的信息，笔者采用的方法是通过Chrome的DevTools来获取，该工具可能需要Webview打开Debug开关。



提示 笔者在测试腾讯地图时，使用的是Android 4.4.4的手机，默认可以通过Chrome的DevTools查看Webview的内容。如果读者发现DevTools不能Debug Webview时，可以尝试在被测程序的WebView中加上代码把Debug打开，代码如下：

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {  
    WebView.setWebContentsDebuggingEnabled(true);  
}
```

先将测试手机设置-开发者选项-Android调试开关打开，再将手机与电脑连接上。然后打开电脑上的Chrome浏览器，在地址栏输入“chrome: //inspect/#devices”，按回车键。此时在浏览器上应该就可

以看到如图6-16所示的画面。如果此时测试手机画面上有可识别的Webview，在该画面上就会显示网页的链接，点击下面的inspect画面，在新的DevTools中查看Webview的内容（图6-17中左图），选择DevTools的某个Element在手机上会同步选中对应的Element（图6-17中右图），右键选中DevTools中的Element还可以复制CSS和XPath。

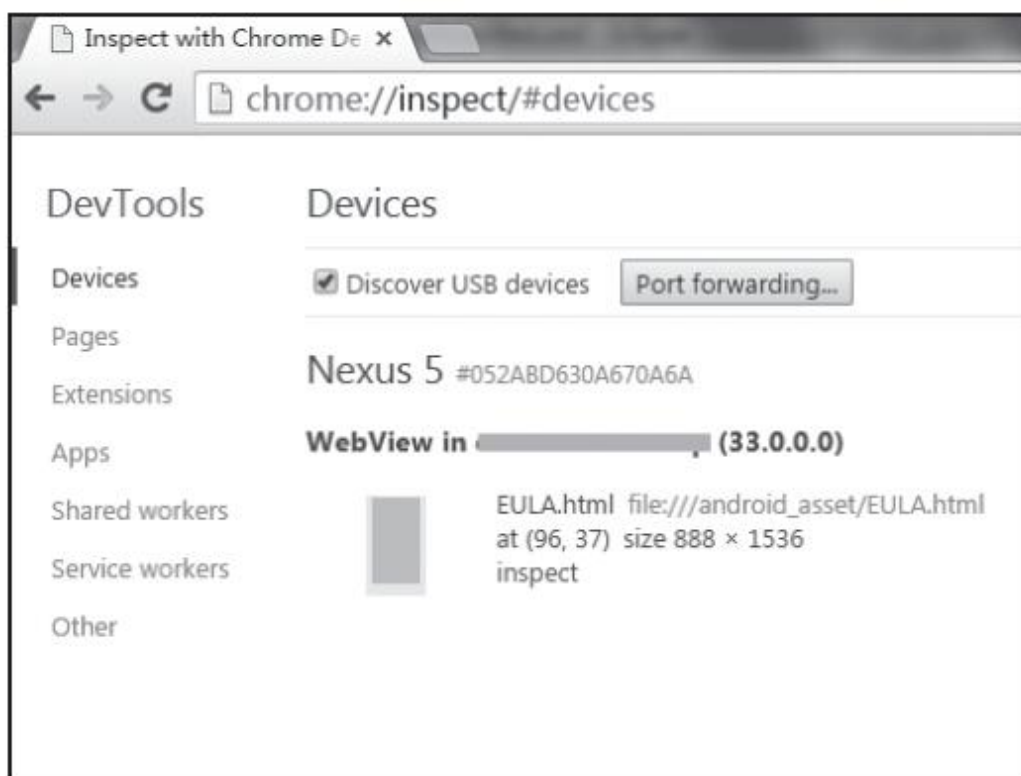


图6-16 Chrome DevTools连接画面

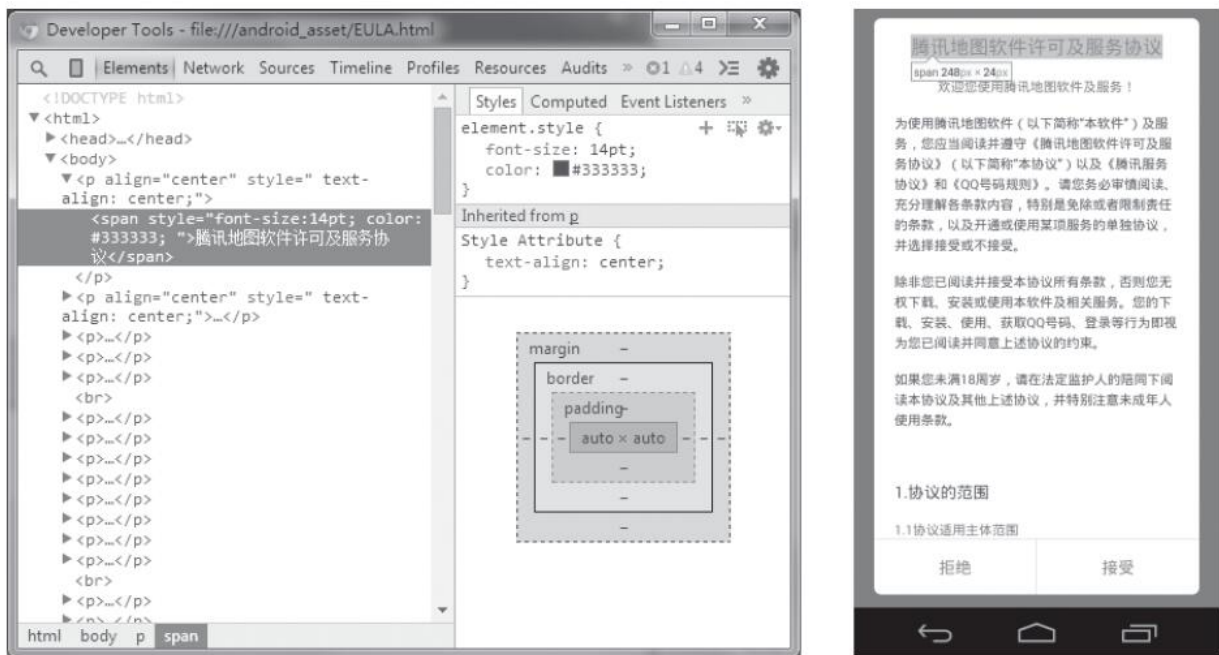


图6-17 DevTools上inspect网页内容

2.不能获取Webview的Context

在测试Hybrid App时可能会遇到Webview的Context不能正确获取的情况，从Appium日志中可以看到Context只有“NATIVE_APP”，而没有Webview的Context，情况如下面的日志所示。

```
info: [debug] Getting a list of available webviews
info: [debug] executing cmd: C:\android\platform-tools\adb.exe -s 052abd630a670a6a
shell "cat /proc/net/unix"
info: [debug] Available contexts:
info: [debug] []
info: [debug] Available contexts: NATIVE_APP
```

出现这种情况的原因可能有以下两种：

首先，Desired Capabilities中automationName使用Appium，但是测试手机不是Android 4.4以上的版本。由于4.4以下版本的Webview没有使用Chrome内核，只有在Selendroid模式中才支持4.4以下的内核。

其次，程序中使用的Webview不是Chrome内核。由于Appium只支持Chrome内核，如果Webview使用的是其他内核，Appium就不能获取该WebView的Context。例如，腾讯地图某些页面就使用了QQ浏览器的X5内核，导致无法测试网页内容。

6.3.5 Appium脚本常见问题及处理方法

本小节我们来看一下Appium脚本常见问题及处理方法。

1.运行时遇到异常“A new session could not be created”

在测试过程中往往会遇到“A new session could not be created”的问题，造成该问题的原因大多数是测试异常退出导致Appium服务器上的Session没有正确结束；另外还可能是被测的应用程序正处在前台，导致脚本不能正确地建立Session。

笔者解决这类问题的方法主要有以下三种：

- 在测试基类的setUp方法中，调用UiHelper的init方法之前用ADB命令强制关闭被测应用一次，保证与Appium连接前被测应用不是在前台运行状态。

- 在Appium的服务器启动时加上“--session-override”，测试脚本在运行过程中同样的Session会覆盖之前的Session，确保Session可以正确创建。

- 在测试基类的tearDown方法中，关闭Appium的Session，每个用例结束时自动关闭Session，下一个用例开始时Appium服务器的环境就是

清洁的环境。

2.在一台电脑上用Appium测试多个手机

一个Appium服务器只能连接一个测试设备进行测试，如果要在同一台电脑上同时测试多个设备，解决方法是在电脑上启动多个Appium服务器，每个Appium服务器分别连接不同的测试设备。由于Appium服务器默认监听4723端口，同时Appium服务器与手机的Bootstrap.jar通信默认使用4724端口，ChromeDriver默认使用9515端口。如不修改启动命令，再次启动Appium时会提示端口已经被占用，因此需要在启动命令中将监听的端口改为未使用的端口，端口号为0~65535。



提示 Appium监听端口和Bootstrap端口是必须指定的，如果仅是测试移动Native应用程序可以不指定ChromeDriver端口。

除了测试服务器监听的端口需要更改以外，还需要分别为每个测试服务器指定连接的测试设备UID。

修改以上信息可以在Appium服务器启动命令中加上以下参数：

·-p: 服务器端口

·-bp: Bootstrap端口

·-U: 连接设备的UID

·--chromedriver-port: ChromeDriver的端口

假设与电脑连接的设备UID分别为1234和2345。首先启动一个命令行窗口，使用默认的端口启动第一个服务器连接到1234的手机，命令如下面的代码所示。

```
node appium.js --session-override -  
p 4723 -  
bp 4724 --chromedriver-port 9515 -  
U 1234
```

再启动另外一个命令行窗口，用非默认的端口启动第二个服务器连接到2345的手机，命令如下面的代码所示。

```
node appium.js --session-override -  
p 4725 -  
bp 4726 --chromedriver-port 9516 -  
U 2345
```

测试服务器已经启动了，此时测试脚本的配置文件需要将连接的端口修改成对应服务器的端口（图6-18），测试脚本使用该配置文件就能连接到第二个测试服务器了，并在第二个手机上进行测试。

```
platformName=Android
platformVersion=4.4.4
udid=052abd630a670a6a
appPackage=com.tencent.map
appActivity=.WelcomeActivity
unicodeKeyboard=true
newCommandTimeout=150
remoteHost=http://127.0.0.1:4725/wd/hub
```

图6-18 与第二个服务器连接的配置文件

3.通过findElementByUIAutomator方法查找列表项

在测试过程中，可能遇到需要查找列表中的某个列表项的问题，但是该列表项具体的位置未知，可能显示在屏幕上，也可能需要滑动列表才能显示出来。例如，如图6-19所示，离线地图数据的城市列表包含全国300多个城市的入口，该列表非常长。如果要找到哈尔滨市，则测试脚本需要向下滑动列表。在不同屏幕分辨率的手机上，滑动屏幕的次数可能是不一样的，因此测试脚本中不能写固定的滑动次数，而应该使用一种更加灵活的方式。



图6-19 腾讯地图离线地图列表

经过笔者的一些尝试，发现通过WebDriver提供的 `find_element_by_android_uiautomator` 方法，可以比较方便地处理这种情况。如下面的代码所示，方法中调用了 `find_element_by_android_uiautomator`，传入的参数是一个字符串，该字符串符合UiAutomator控件查找的Java语法。在本示例中，脚本首先通过判断控件可以滑动的属性找到了一个UiScrollable的对象，然后在UiScrollable的对象中根据Child控件的类型 `android.widget.LinearLayout`

和显示的文本信息查找子控件。这种方法的优点是脚本不需要知道列表项的具体位置，更加通用。脚本在运行时会自动上下滑动列表，直到找到满足条件的列表项为止。更多关于使用UiAutomator的信息请参考第7章的内容或者登录Android开发者网站查询。

```
def getListItemByText(cls, searchText):  
    return cls.uiHelper._driver.find_element_by_android_uiautomator('new  
    UiScrollable(new UiSelector().scrollable(true)).getChildByText(new UiSelector().  
    className(android.widget.LinearLayout), "%s")' %searchText)
```

6.4 本章小结

本章先概要介绍了Appium框架的原理、优缺点以及如何判断该工具是否适用于项目的自动化测试，然后阐述了如何搭建Appium的测试环境，并结合Android系统自带的联系人功能讲述了如何利用Python语言写一个Appium的自动化测试脚本。接下来结合笔者在腾讯地图项目中的实践经验，阐述了在实际项目中如何高效地利用Appium实施自动化测试和在实施过程中经常遇到的一些问题的原因与解决方法。希望读者能在阅读本章后，对Appium有一个较深的理解，能将Appium和笔者的经验应用到项目中去。

第7章 Android App速度测试

移动互联网在快速发展，App的竞争呈现白热化。App的性能表现不再是可有可无的指标，而是影响用户选择或者放弃一款App的重要依据。以手机浏览器为例，CNNIC 2013年发布的一份《中国手机浏览器用户研究报告》显示，手机浏览器自身的性能和速度是影响用户选择的最重要因素，如图7-1所示。

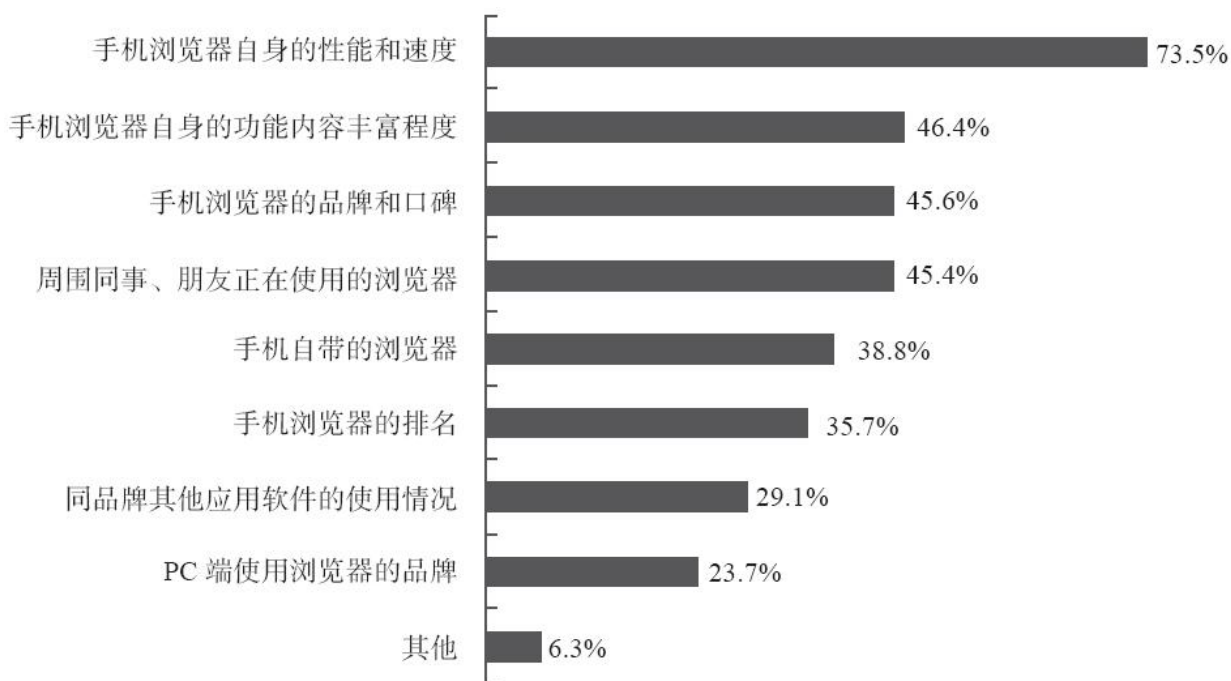


图7-1 影响用户选择手机浏览器的因素

一旦性能出现问题，用户很可能会因此而流失。据统计，当App网页打开时间超过2000ms时，用户开始流失；当App交互执行性能时间达到400ms时，性能开始出现隐患。

本章主要介绍在众多性能测试项中扮演最重要角色的速度测试。第一部分会列举影响用户体验的速度测试场景。第二部分引出速度测试的多种方法，并介绍这些方法的优缺点。第三部分以两个实例详细说明速度测试方法在手机浏览器项目中的实践过程。第四部分总结速度测试的测试心得。本章知识结构图如图7-2所示。

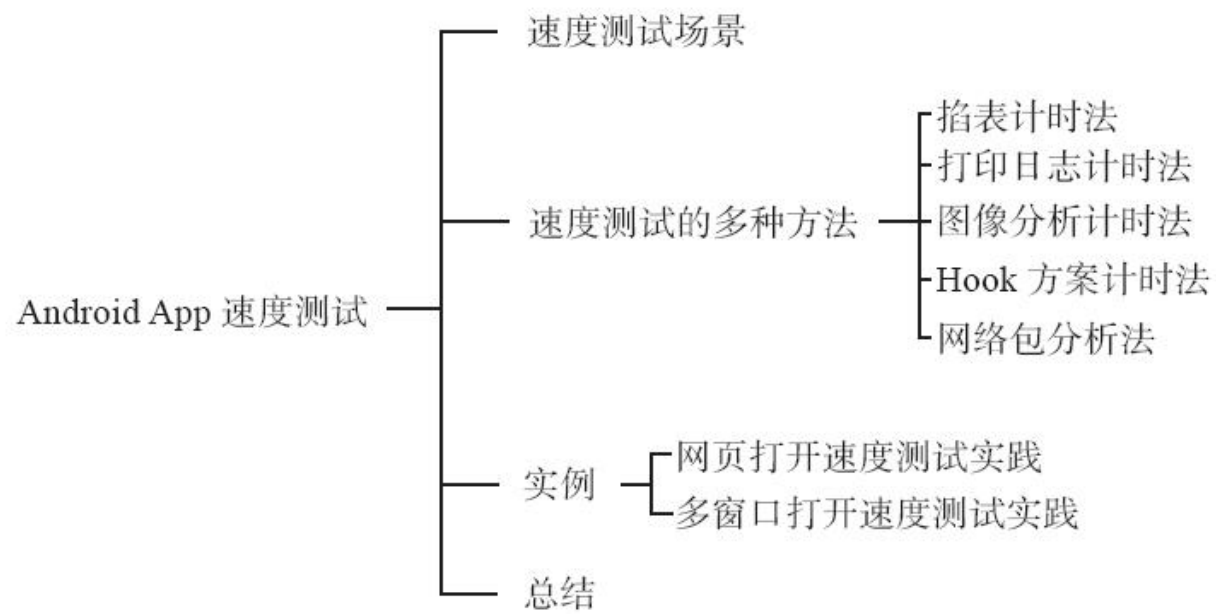


图7-2 本章知识结构图

7.1 速度测试场景

哪些场景需要做速度测试？笔者认为可以遵循以下两个原则：

1.重要性原则

重要性原则的意思是，越重要的场景就越有必要去做速度测试。一个场景的重要性可以通过用户对该场景的使用频率以及当前版本的重点功能来评估。以手机浏览器为例，通过用户数据收集可以得知：浏览器启动、退出、搜索栏点击这三个操作的日使用用户数都在100万以上，所以笔者认为这三个操作的速度需要重点保证（表7-1）。

表7-1 手机浏览器各功能日使用用户数（示意数据）

功能项类型	功能项	功能日使用用户数
内核	启动	1983471
菜单	菜单－退出	1700292
地址栏	搜索栏－点击次数	1121472
(续)		
功能项类型	功能项	功能日使用用户数
菜单	菜单－书签	482220
多窗口	多窗口－点“×”关闭	479898
...

2.痛点原则


痛点原则的意思是，越是让用户感觉难受的、抱怨的速度问题就越有必要做速度测试。通常可以从用户反馈信息和用户调查问卷中，了解到相关信息并做出选择。还是以手机浏览器为例，有不少浏览器反馈新建多窗口慢，测试组对新建多窗口场景进行了速度测试。

测试场景具有多方面属性，除了需要选择测试动作之外，还需要指定其他属性。比如网络、机型、是否有缓存、是冷启动还是热启动等。手机浏览器测试团队为了优化弱网络情况下的网页打开速度，就曾经进行过多次外场测试。选取地铁、公交、麦当劳等用户常见的弱网络场所进行网页打开速度测试，并有针对性地优化网页打开速度。

表7-2是手机浏览器选取的速度测试场景，供大家参考。

表7-2 五种常见的速度测试场景

速度测试场景	描述	选取原因
启动速度	用户点击手机浏览器图标到起始页出现的时间	用户必经操作
多窗口操作速度	用户点击多窗口到多窗口界面出现的时间	用户反馈性能问题重灾区
打开网页速度	打开一个网页的速度	手机浏览器的最核心操作
下载速度	下载一个 App 的速度	基础功能
视频打开速度	点击播放一个视频到视频出现第一帧的时间	重点推广功能



名词解释

冷启动与热启动

Android系统的Activity退出之后，应用的进程并不会被“杀死”，而是保留在那里。当再次打开App的Activity时，会从已有的进程中创建

Activity，是为“热启动”。若打开Activity时没有进程，则会先创建一个进程，再在新建的进程中打开Activity，是为“冷启动”。

7.2 速度测试的六大方法

选定了测试场景之后，就马上开始测试。做速度测试不需要高深的技术和专门的设备，简单一个秒表（智能手机都有秒表功能）、一支笔、一部手机就可以开始测试了。操作过程也很简单，以手机浏览器启动速度为例，步骤如下：

（1）确保手机已经安装手机浏览器，并且未启动。

（2）活动桌面让手机浏览器图标处于当前桌面。秒表清零。

（3）点击手机浏览器图标的同时按下秒表，开始计时。

（4）手机浏览器自动跳过欢迎页显示起始页（启动完成）。此页面出现立马按停秒表，计时结束。

（5）记录秒表时间。

（6）重复（1）~（5）步多次，获得多次测试结果取平均值。

以上就是一个最简单的速度测试过程。这个过程可以分成以下三部分。

1.操作手机

操作手机最简单的方法就是手动点击屏幕。这种方法优点是简单、灵活，对不同手机的兼容性高；而缺点是容易点错、操作效率不高、大量重复操作容易疲劳。所以手动点击屏幕的方法通常用于初次摸底测试，以便对被测速度指标有一个大致的了解。当需要常规测试时，通常会选用自动化方式操作手机。本书在其他章节中对Monkey、UIAutomator、Robotium等自动化工具的使用方法已有介绍，这里不再赘述了。

2.记录测试结果

记录测试结果指的是识别被测操作开始和结束的时间并记录下来。上例中，识别开始和结束的时间使用的是人眼，并通过纸笔来记录该时间。和手动点击屏幕方法一样，人眼识别方法简单、灵活，但不适用于大量的重复测试，也无法保证精确度。所以需要引入一些自动化的识别方法，包括打印日志计时法、图像分析计时法、Hook方案计时法、网络包分析法等。这些方法正是速度测试研究的主要技术，本章后面会详细展开介绍。

3.对测试结果进行数据处理得到测试值

对测试结果进行数据处理得到测试值指的是通过一组测量值得出一个平均值的过程。因为每次测试时的手机状态、网络状态会有不同，同一个操作多次测量的结果也会有偏差，所以通常需要以测试多

次取平均值的方法来减少误差。如何去除粗大误差、如何取平均值、如何评估结果的误差属于误差统计学的研究范畴，本书不准备讨论这部分内容。但笔者会在下面案例介绍环节中将有用的经验介绍给大家。

下面开始逐一介绍各种速度测试方法以及它们的使用场景。

7.2.1 掐表计时法

前文已经提到，掐表是大家最常见、最容易想到的计时方法，如图7-3所示。操作的同时开始计时，预期结果展示完成时结束计时，其间所花费的时间就一目了然了。这种方法特别适合测量指标时间长（大于10s）、对测试精度要求较低的临时性测试。但是如果软件的操作都是短短几秒或者1s内完成的动作，用这种方式产生的误差就显得太大了，会直接导致测试结论的错误。



图7-3 掐表计时法

也就是说，通过测试时间、精度要求和测试频率这三个条件可以判断一项速度测试是否适合用掐表计时法。以手机浏览器选定的五种典型测试场景为例，下载速度场景测试时间长（10~50s）、精度要求低（500ms）、测试频率低（每个版本一次），所以适合用掐表计时法测试。而启动速度、多窗口操作速度、网页打开速度和视频打开速度的精度要求都在50ms以下。如表7-3中的场景用掐表计时法就无法满足要求。

表7-3 五种典型测试场景的精度要求

速度测试场景	测试时间	精度要求	测试频率
启动速度	1 ~ 2s	50ms	每个版本测试
多窗口操作速度	200 ~ 500ms	50ms	每个版本测试
网页打开速度	1000 ~ 5000ms	50ms	每天 DailyBuild 测试
下载速度	10 ~ 50s	500ms	每个版本测试
视频打开速度	1 ~ 3s	50ms	每个版本测试

有没有更精确一点儿的方法呢？来看看打印日志计时法吧！

7.2.2 打印日志计时法

这个其实很好理解，就是在关键节点通过接口打印出有用的日志，分析这些日志即可得出开始和结束的时间。比如，假设执行某个操作时调用的第一个接口代码如下：

```
startUp(){  
    //.....  
  
}
```

预期结果显示完成时调用的接口代码如下：

```
endUp(){  
    //.....  
  
}
```

那么在编码阶段，可以在这两个接口中分别打印当前时间，如下面的代码所示。

```
startUp(){  
    print(os.time())  
    //.....  
  
}  
endUp(){  
    //.....  
  
    print(os.time())  
}
```

这样在执行一次这个操作时，就会打印两个时间节点，这两个时间节点能够精确地计算这个操作完成的时间。

具体以网页打开速度为例，在开始打开网页时浏览器会调用一个叫**onStart**的函数。而在网页打开完成时，浏览器会调用一个叫**onFinish**的函数。这样，只要在这两个接口中打印出当前时间，在执行打开网页操作时，就会打印开始和完成的时间节点。这两个时间节点能够精确地计算打开网页完成的时间。除此之外，还能打印出一些过程函数的耗时，这样便于分析程序慢在哪里，是非常不错的一个方式。

但这种方式也有一些不足之处。首先，测试前需要源码，而竞品的源码不可能得到，这样就无法与竞品对比。其次，日志打印时间和用户感知时间可能有偏差。以网页的打开时间为例，程序认为**onFinish**函数执行时首屏就出来了，但渲染和上屏的耗时程序无法获知，用户感知到的首屏会比**onFinish**的打印时间要晚，而通常项目组更关注的是用户感知到首屏的显示时间。

有没有能对比竞品，并且更能体现用户感知的速度的测试方法呢？来看看图像分析计时法！

7.2.3 图像分析计时法

图像分析计时法的大体思路是用工具记录操作过程每一时刻的屏幕图像和图像对应的时间，然后通过分析算法找出开始和结束的图像，从而获得开始和结束的时间，如图7-4所示。

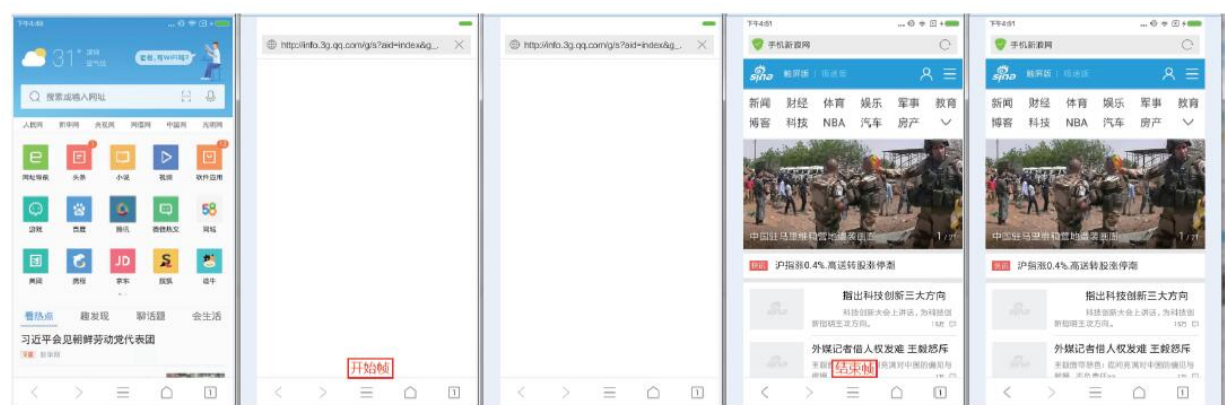


图7-4 通过算法找出开始帧和结束帧

这种方法理论上和人眼的感知一致，而且不需要修改程序源码。准确性则要看使用什么工具来获取图像。一般都能达到每秒30帧或以上的频率，也就是说精度能控制在33ms以内。获取图像的工具有的直接生成图片，有的生成的是视频；有的是测试手机上安装的App，有的则需要借助别的设备来完成图像获取。根据特性的不同可以分为以下四种：截屏、录屏、拍照、录像，见表7-4。

表7-4 获取图像的四种方式

	手机 App	借助其他工具
获取图片	截屏	拍照
获取视频	录屏	录像

这四种方式的具体工具推荐以及这些工具的优缺点见表7-5。大家可以根据自身情况选择适合自己的。

截屏和拍照得到的图片可以直接分析开始和结束的时间，视频则需要先经过分帧成为图片才能分析开始和结束的时间。同一个测试场景要测试多次取平均值，而且每次测试需要处理很多分帧图片。人工找起来很费劲，所以智能分析算法必不可少。通过图片分析开始和结束的时间的算法成为图像分析方法的关键。

表7-5 获取图像的四种方式的优缺点

记录方式	推荐工具	优点	缺点
截屏	使用 Android 源代码中的 capture-Screen 接口 详细使用方法参见 8.3.4 节 帧率：16 帧 / 秒	1. 查看图片即可直接计算时间（每一张图片都有生成时间） 2. 不需要另外的设备 3. 图片清晰好处理（可以真实记录屏幕像素）	1. 全程高速截屏用占用非常多的手机系统资源，可能影响被测软件本身的性能 2. 图片保存在手机上占用手机存储空间，需要及时复制至 PC
录屏	Android 4.4 版本后自带录屏工具 adb shell screenrecord 帧率：60 帧 / 秒	同截屏方式	1. 需要先分帧才能计算时间，相对麻烦 2. 其他同截屏方式
拍照	暂未发现好用的工具	1. 不需要占用手机系统资源 2. 不用分帧直接通过图片就能计算时间（每一张图片都有生成时间）	1. 需要另一台设备（如相机），记录过程容易出错 2. 图片不清晰（光线、对焦、拍摄角度等都对成像有影响）
录像	罗技 C920 网络摄像头 + 罗技摄像头软件 Logitech Webcam Software 帧率：30 帧 / 秒	不需要占用手机系统资源	1. 需要另一台设备，记录过程容易出错（一种出错是摄像头软件卡死；另一种出错是录像操作与手机操作不同步） 2. 图片不清晰（光线、对焦、拍摄角度等都对成像有影响） 3. 需要先分帧才能计算时间，相对麻烦

其基本思路其实很简单，就是找到开始帧和结束帧的特征，并通过一定的图像算法识别到这个特征。但实际操作起来却会遇到不少问题，下面尝试逐一解决这些问题。

1.识别开始帧

开始帧就是打开网页时点击“前往”按钮那一刻的图片，又或者是下载应用点击“下载”按钮那一刻的图片。这听起来似乎很简单，但事实上这个时间点在屏幕上有时是没有显示的。比如点击“前往”按钮，程序可能需要执行一些逻辑后才让“前进”按钮显示为点击态，这样就没办法准确找出开始帧。

那怎么办？

几经探索，笔者终于找到了解决办法：在界面上留下操作的痕迹。比如点击时在点击位置绘制一个白点，滑动时绘制滑动的轨迹。这个白点需要打开“系统设置→开发者选项→显示触摸操作”才会显示出来。设置界面如图7-5所示。设置后效果如图7-6所示。



图7-5 设置“显示触摸操作”界面



图7-6 设置“显示触摸操作”后效果

测试数据表明，点击事件发生到白点显示之间的延时小于30ms，小于录像的分帧间隔。

白点识别的算法会在实例章节中予以介绍，这里就不展开叙述了。

2.识别结束帧

结束帧就是测试场景动作完成的那一帧，比如打开网页场景中网页铺面屏幕，又比如下载场景中显示下载完成。对于动作完成界面不变的情形，结束帧很好找，直接用图像对比方法就能找到。

如图7-7所示，网页打开之后就不再变化了。这时可以将最后一张图片作为结束帧标准图片，然后从前往后将每个图片依次与这个标准图片进行对比。当图像对比一致时，则认为找到了结束帧。



图7-7 网页打开后页面不再变化

但如果动作场景完成后，界面还会变化，比如手腾网打开后，大图幻灯片还在来回切换，这种情况就变得复杂了，这时就需要采取局部图像对比等更有针对性的方法来找结束帧。详情见实例。

看到这里，想必大家也看出了图像分析方法的缺点：分析操作烦琐、实现自动化难度大、有出错概率需要人工校验。

那么还有更好的方案吗？

7.2.4 Hook方案计时法

对于速度的感知，从人的角度讲，以按下手指，人感知为开始；到菜单完全弹出，人感知为结束，那么这两个点是否可以对应到程序里？当然可以！从程序的角度讲，按下手指，即系统收到触摸消息，菜单完全弹出，则可以对应到菜单绘制结束，如图7-8所示。

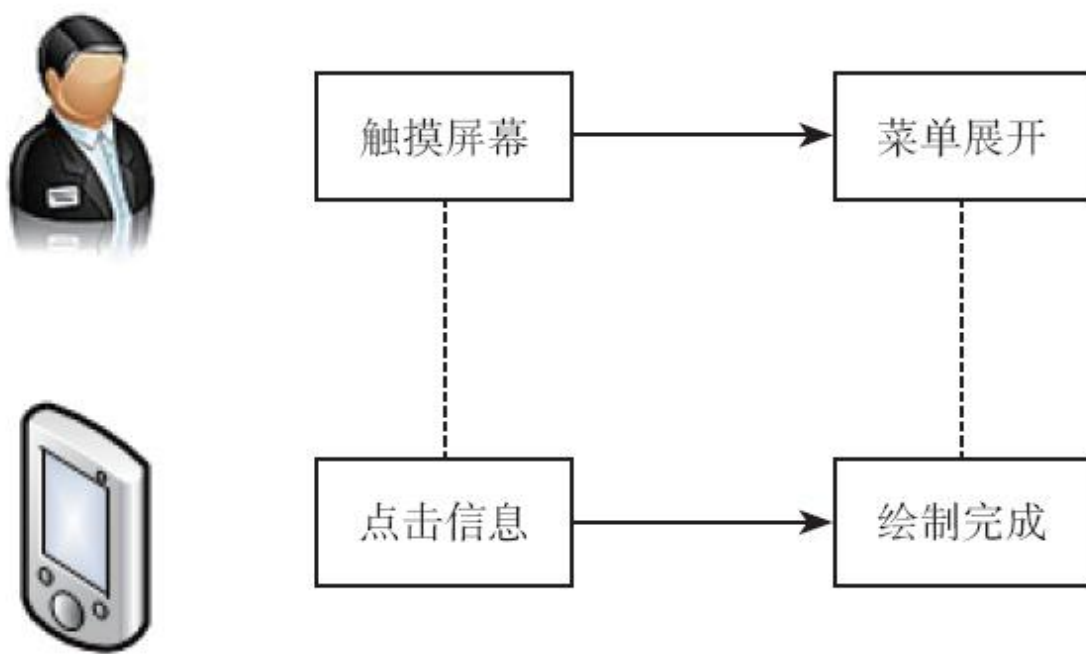


图7-8 速度的定义

如何在程序里获取点击消息和绘制完成这两个时间节点？

获取点击消息，考虑Android点击消息分发流程，如图7-9所示。

Android程序的用户点击消息处理，一般是由View来完成的。如果可以在View.dispatchTouchEvent或者View.onTouchEvent这两个函数执行前记录下时间（比如打个log），那么就可以获取用户触摸屏幕的时间。如果View被添加了onTouchListener，那么View.onTouchEvent函数将不会执行，所以，通过hook View.dispatchTouchEvent记录用户点击消息的时间。

获取绘制完成，先来了解Android绘制的大概流程，如图7-10所示。

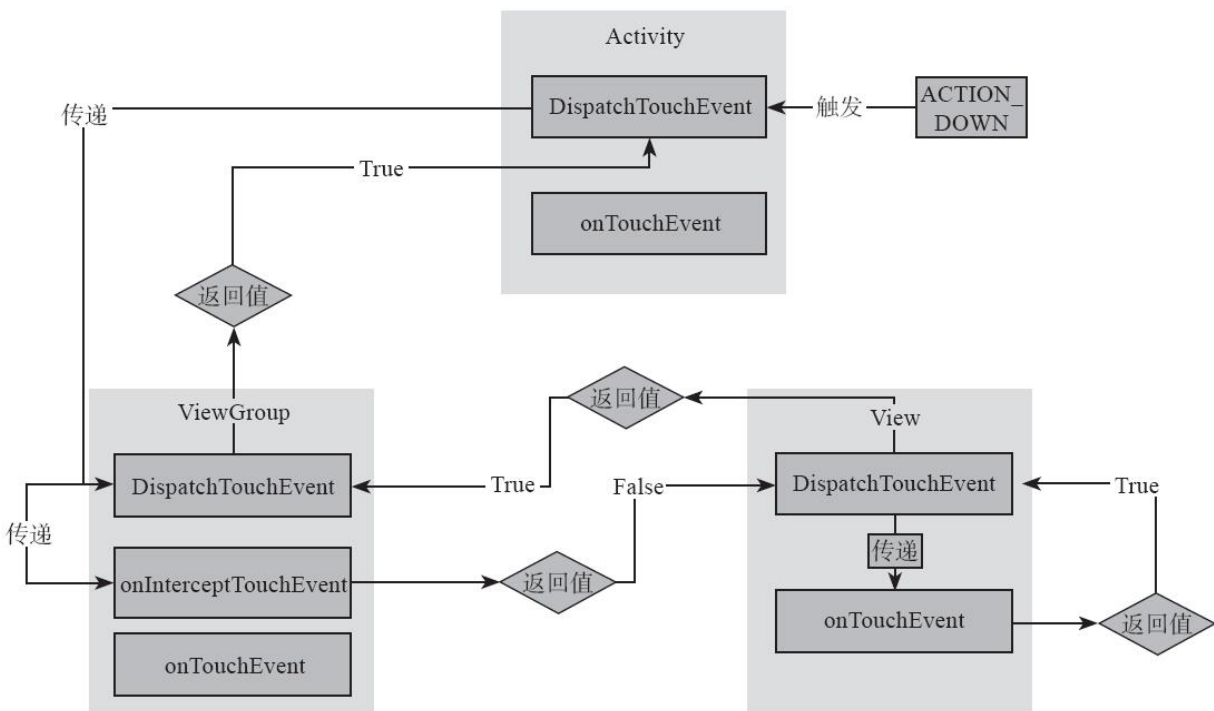


图7-9 Andriod点击消息分发流程

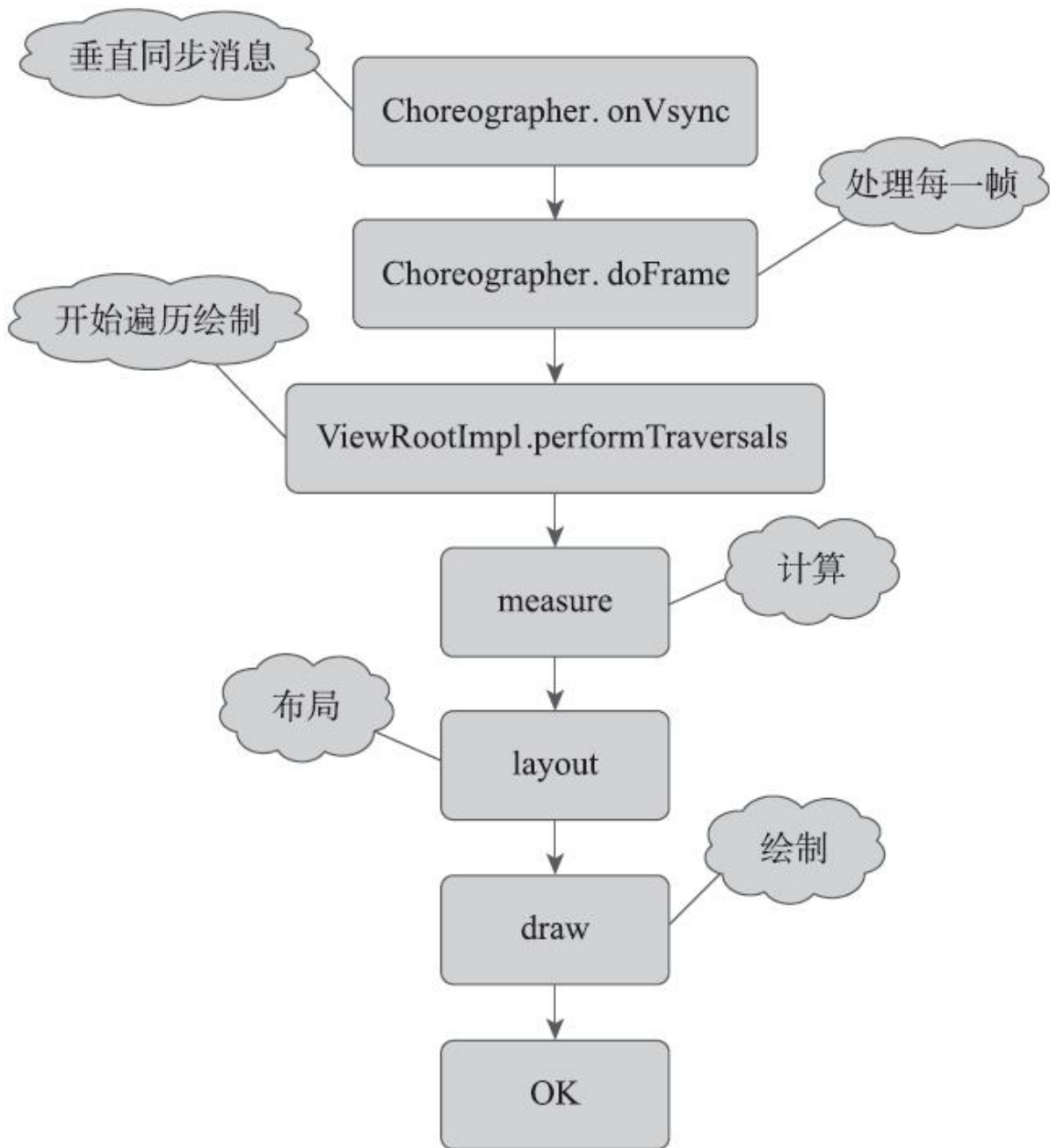


图7-10 Android绘制流程一

Android控件的绘制，从`ViewRootImpl.performTraversals`开始，函数将遍历每个view，根据需求依次进行`measure`（计算）、`layout`（布局）、`draw`（绘制）的过程。`View.draw`函数是控件真正绘制的函数。

继续跟踪View.draw，由于现在的4.X手机基本都支持硬件加速，所以View.draw最终是调用GPU进行绘制的，也就是调用android.view.Renderer\$GLRenderer.draw。在这个函数里面，程序分三步完成了控件的绘制，如图7-11所示。

- (1) CPU遍历每个控件，录制绘图命令，生成build DisplayList。
- (2) GPU回放draw DisplayList，将图形绘制到buffer上。
- (3) 交换buffer，真正在屏幕上显示出来。

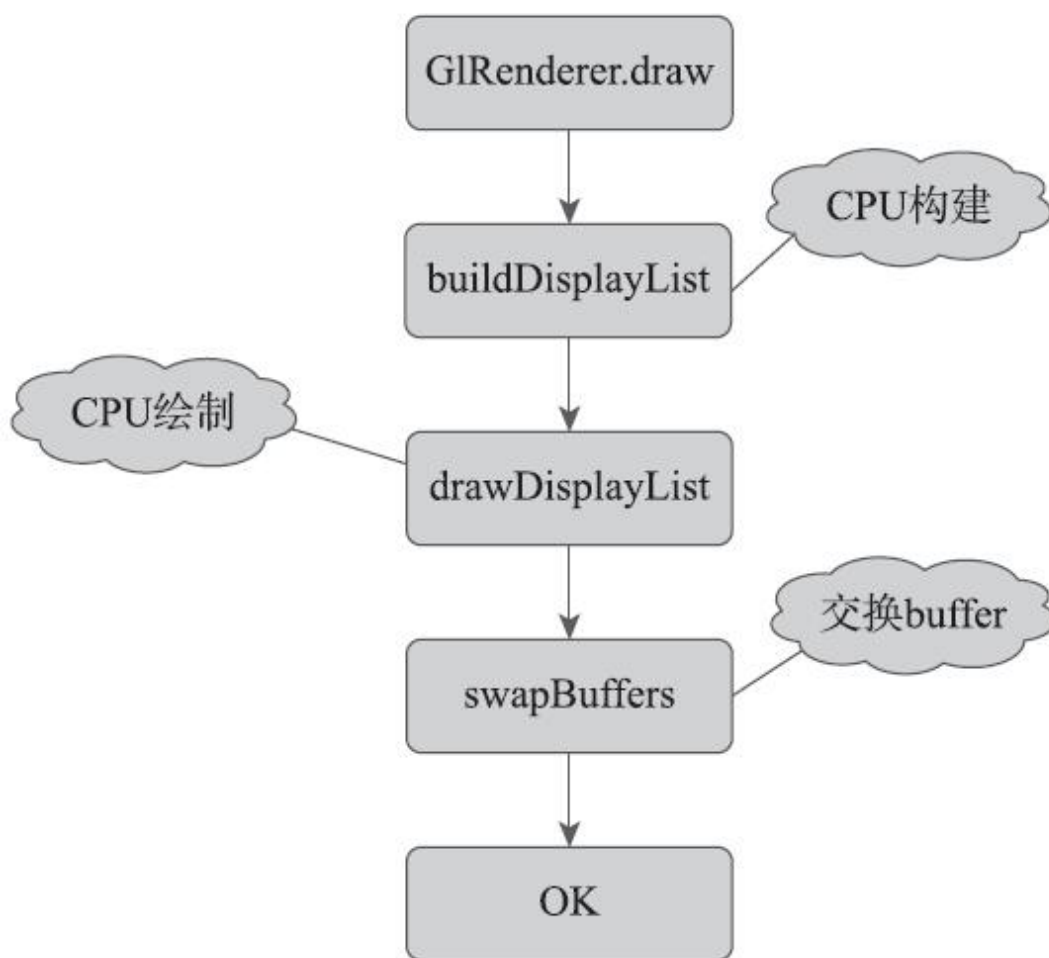
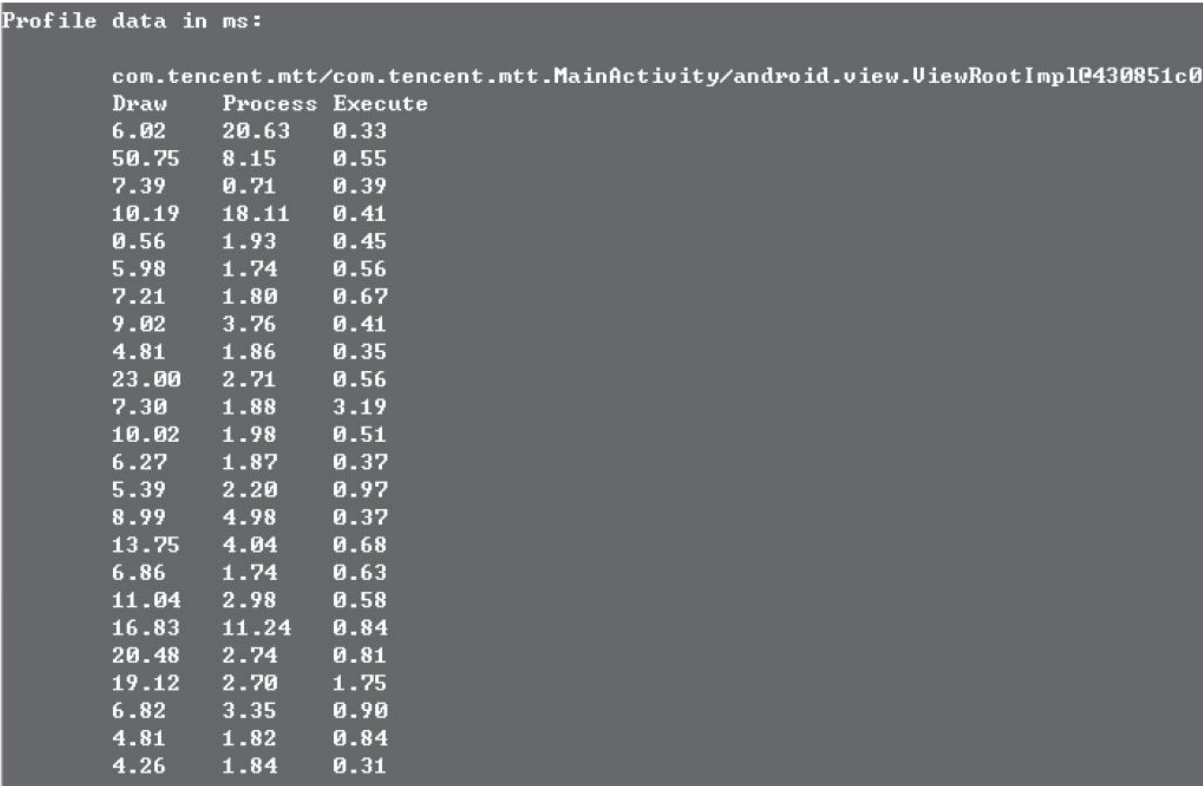


图7-11 Android绘制流程二

在Android手机上，如果打开“开发者选项→GPU”呈现模式分析，系统就会对上述三个函数进行计时，可以采用adb shell dumpsys gfxinfo看到App的每一帧的详细绘制情况，如图7-12所示。

其中，Draw统计的是buildDisplayList耗时，Process统计的是drawDisplayList耗时，Execute统计的是swapBuffers耗时。

经过上面的分析可知，Android系统是根据GLRenderer.draw函数来调试控件绘制的，所以可以执行hook GLRenderer.draw，如果这个函数执行完毕，则确定为当前帧绘制完毕。



```
Profile data in ms:
com.tencent.mtt/com.tencent.mtt.MainActivity/android.view.ViewRootImpl@430851c0
Draw    Process Execute
6.02    20.63    0.33
50.75   8.15     0.55
7.39    0.71     0.39
10.19   18.11    0.41
0.56    1.93     0.45
5.98    1.74     0.56
7.21    1.80     0.67
9.02    3.76     0.41
4.81    1.86     0.35
23.00   2.71     0.56
7.30    1.88     3.19
10.02   1.98     0.51
6.27    1.87     0.37
5.39    2.20     0.97
8.99    4.98     0.37
13.75   4.04     0.68
6.86    1.74     0.63
11.04   2.98     0.58
16.83   11.24    0.84
20.48   2.74     0.81
19.12   2.70     1.75
6.82    3.35     0.90
4.81    1.82     0.84
4.26    1.84     0.31
```

图7-12 Dump绘制信息

到目前为止，已经确定了两个关键函数：

- (1) `View.dispatchTouchEvent` → 获取用户点击时间。
- (2) `GLRenderer.draw` → 获取绘制完成时间。

通过计算这两个时间差，就可以得到从用户点击菜单键到菜单弹出来的时间。

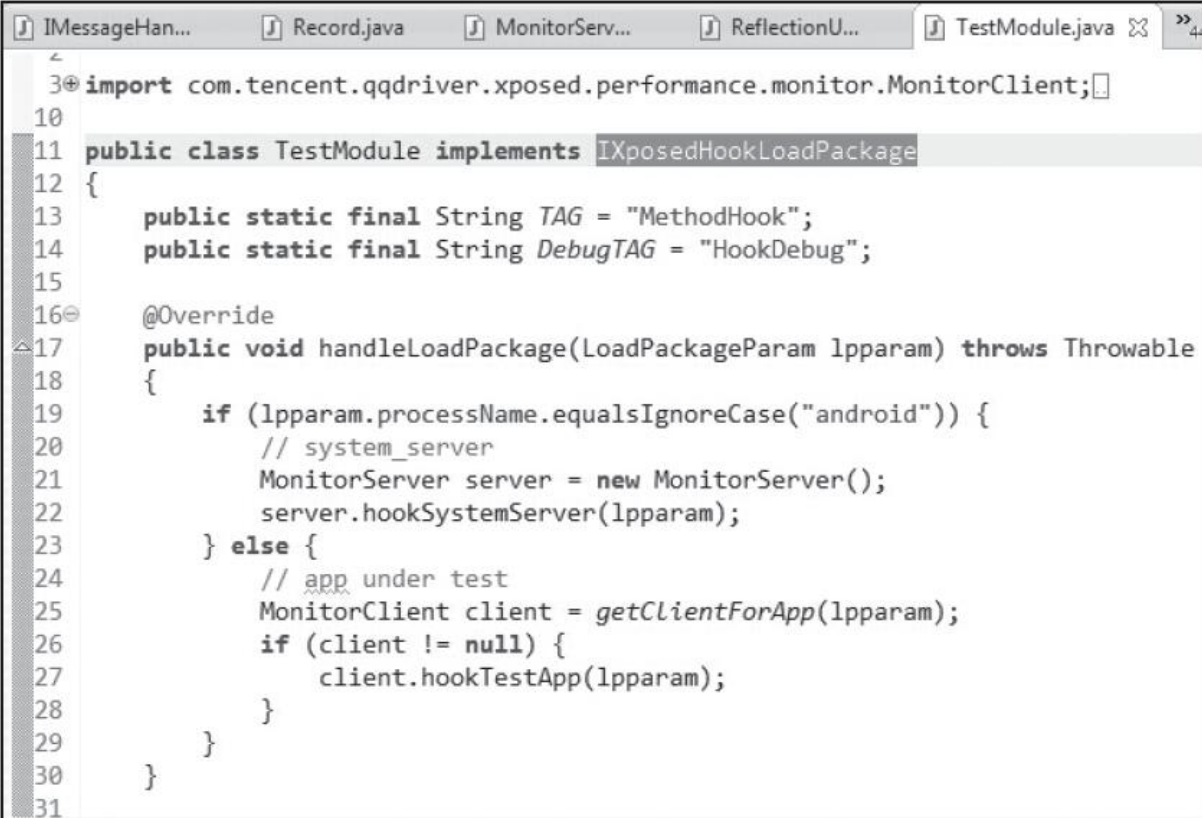
熟悉了原理，通过Xposed框架来实现具体的Hook代码编写。

Xposed框架是一个Android系统扩展库，安装了这个框架以后，有各种各样的插件供用户使用。这些插件可以任意修改系统，小到修改下拨号界面背景色，大到修改整个拨号界面，都可以实现。

Xposed框架支持4.X主流系统，当然还是需要ROOT的手机。

要完成一个Xposed插件，需要引入Xposed的jar包，并且实现IXposedHookLoad-Package接口，如图7-13所示。

需要hook一个Java函数非常简单，只需要传入Java类全名，方法名，参数类型，其他则交由Xposed框架处理。



```
10
11 import com.tencent.qqdriver.xposed.performance.monitor.MonitorClient;
12
13 public class TestModule implements IXposedHookLoadPackage
14 {
15     public static final String TAG = "MethodHook";
16     public static final String DebugTAG = "HookDebug";
17
18     @Override
19     public void handleLoadPackage(LoadPackageParam lpparam) throws Throwable
20     {
21         if (lpparam.processName.equalsIgnoreCase("android")) {
22             // system_server
23             MonitorServer server = new MonitorServer();
24             server.hookSystemServer(lpparam);
25         } else {
26             // app under test
27             MonitorClient client = getClientForApp(lpparam);
28             if (client != null) {
29                 client.hookTestApp(lpparam);
30             }
31         }
32     }
33 }
```

图7-13 Xposed Hook实现

7.2.5 网络包分析法

网络包分析法不是一种通用的速度测试方法，因为有的速度测试场景不涉及网络交互。同时，它也不能准确度量某个场景的用户感知速度，因为网络包传输完网页不一定就能显示出来。但是，它确实是发现程序慢在哪里的重要方法。

首先，需要抓取网络包。在PC端，可选的抓包工具比较多，包括Wireshark、HttpWatch等。而在手机端，推荐使用Tcpdump。

Tcpdump使用方法如下：

步骤1：手机要有ROOT权限

步骤2：下载Tcpdump

步骤3：adb push c: \wherever_you_put\tcpdump\data/local/tcpdump

步骤4：adb shell chmod 6755/data/local/tcpdump

步骤5：adb shell，su获得ROOT权限

步骤6：cd/data/local

步骤7：../tcpdump-p-vv-s 0-w/sdcard/capture.pcap

步骤8: `adb pull/sdcard/capture.pcap d: /`

通过以上八个步骤，网络包就能成功抓取并从手机复制到PC的D盘根目录下，是一个.pcap格式文件，需要借助Wireshark等工具才能打开查看。



注意 关于Tcpdump更详细的信息和工具下载可以访问以下两个网站：

Tcpdump介绍: <http://baike.baidu.com/view/76504.htm> 。

Tcpdump下载: <http://www.tcpdump.org/#latest-release> 。

接下来看看如何分析一个网络包。

以网页打开场景为例，通过网络包分析可以剖析主资源以至每个子资源的连接、请求、等待、接收每个环节的耗时，这对找到程序慢的原因以便有针对性地优化无疑作用巨大。

图7-14所示为通过<http://pcapperf.appspot.com/> 在线工具分析一个网络包的结果。

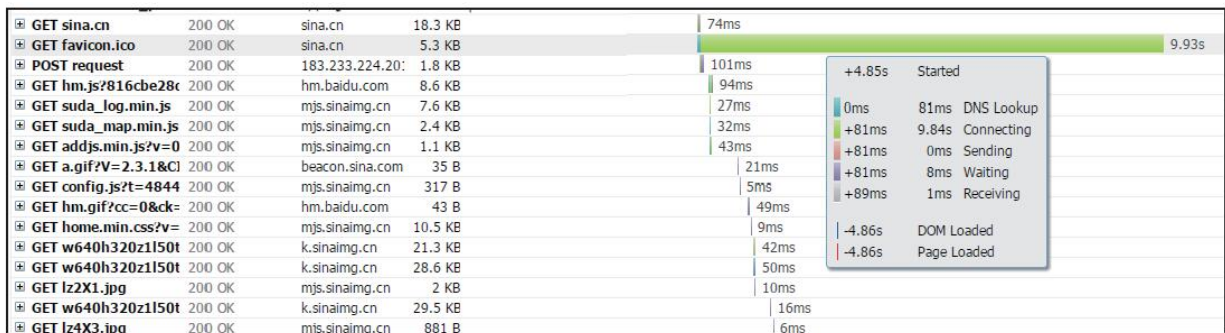


图7-14 在线工具分析网络包的结果

从图7-14中可以看出sina.cn是这个网页的主资源，大小是18.3KB，资源请求总耗时74ms；favicon.ico是sina.cn的一个子资源，大小是5.3KB。从4.85s开始发起请求，DNS查询耗时81ms；连接服务器耗时9.84s；请求包传输耗时0ms；等待服务器回应耗时8ms；回包传输耗时1ms。

除此之外，还可以看到子资源列表中有一些js、jpg、gif等资源。

通过这些资源的请求发起时间和连接、传输时间，开发人员就基本可以找到网络测试慢的原因了。

7.2.6 各种速度测试方法的优缺点

下面总结一下五种速度测试方法的效果和优缺点，见表7-6。

表7-6 五种速度测试方法的效果和优缺点

方法	方法说明	达到测试目的	优点	缺点
掐表计时法	使用秒表。分别记录被测操作开始和结束的时间	度量	1. 简单 2. 真实	1. 精度低，误差高达0.5秒 2. 没法实现自动化
打印日志计时法	分别在被测操作开始和结束的时候打印日志，然后统计时间（包括 App 日志、系统 Logcat、Trace 文件等）	度量 发现慢在哪里	1. 简单、准确 2. 容易实现自动化 3. 方便统计过程指标，找出慢在哪里	1. 日志打印时间和用户感知时间可能有偏差 2. 竞品测试不方便打印日志
图像分析计时法	用录屏或高速摄像头等方式将操作过程录下来。然后通过分帧、调整、统计系列过程得出耗时	度量	1. 中等精度 2. 与用户感知一致	1. 分析视频操作烦琐 2. 实现自动化难度较大 3. 分析结果有出错概率，需要人工校验
Hook 方案计时法	对系统函数插桩，记录时间戳，计算时间消耗	度量	1. 操作完毕立即出结果 2. 高精度	1. 需要 ROOT 手机 2. 需要 Xposed 框架
网络包分析法	通过分析网络包等间接手段计算时间	度量 发现慢在哪里	通过分析网络包发现慢在哪里	测试结果和用户感知可能存在较大偏差

这些测试方法与速度测试的五种场景的适用关系见表7-7。

表7-7 测试方法与测试场景的适用关系

方法	启动速度	操作响应速度	网页打开速度	下载速度	视频打开速度
掐表计时法	不适用	不适用	部分适用	部分适用	不适用
打印日志计时法	部分适用	部分适用	部分适用	部分适用	部分适用
图像分析计时法	适用	适用	适用	适用	适用
Hook 方案计时法	适用	适用	部分适用	适用	部分适用
网络包分析法	部分适用	部分适用	适用	适用	不适用

7.3 手机QQ浏览器网页打开速度测试实践案例

7.3.1 确定关键指标

作为一款工具类的App，手机QQ浏览器将简洁、稳定和快作为重点保证的基础能力。而浏览器打开网页速度的快慢则是用户评价浏览器快慢的关键场景。

但用什么关键指标去衡量浏览器打开网页的快慢，业界并没有统一的标准。手机QQ浏览器测试团队站在以用户体验为中心的角度上，选定了首字时间和首屏时间两个关键指标。

- 首字时间是从用户点击“进入”按钮到手机屏幕页面出现第一个文字或图片的时间。

- 首屏时间是从用户点击“进入”按钮到手机屏幕页面铺满内容的时间。

除了选择关键指标外，还要选择测试的网络类型、测试站点、是否有缓存、是冷启动还是热启动等。由于这些因素对测试方法的影响较小，在这里就不深入讨论了。

7.3.2 选择测试方法

选定了关键指标之后，再选择测试方法。看看哪些测试方法能够度量网页打开速度，哪些方法能够定位慢在哪里。针对当前的测试目标，我们对测试方法做了以下几方面分析。

·**打印日志计时法**：网页打开过程大致可以分为：资源请求—解析—排版—渲染—上屏五个环节，而上屏环节的结束时间程序自身是不知道的。所以打印日志计时法无法准确测量用户感知到的首屏时间。此方法只能作为辅助测试方法分析“慢在哪里”。（其实也可以测试一些过程指标耗时，监控版本间的性能变化）

·**图像分析计时法**：此方法可以测量用户感知的首屏时间，但实现自动化有相当大的难度，而且通过录屏方式会影响手机自身性能（经验证，首屏指标会相差100~300ms，而且不同竞品影响程度不同），只能使用录像方式。另外，测试人员还尝试过使用视频采集卡的方法获取录制手机屏幕，但最终因视频采集卡设备兼容性较差而放弃。录像方法示意图如图7-15所示。

·**Hook方案计时法**：通过前面Hook方案的介绍可知，hook方案适用于结束帧明确的测试场景。而网页打开速度测试结束帧很不明确。首屏出现后，还可能有幻灯片滚动，有广告弹出，因此放弃此方法。

·网络包分析法：通过分析网络包可以分析“慢在哪里”。

综上所述，测试团队选择了图像分析计时法（主要方法）+打印日志计时法（辅助方法）+网络包分析法（辅助方法）作为手机QQ浏览器网页打开速度测试的方法组合。

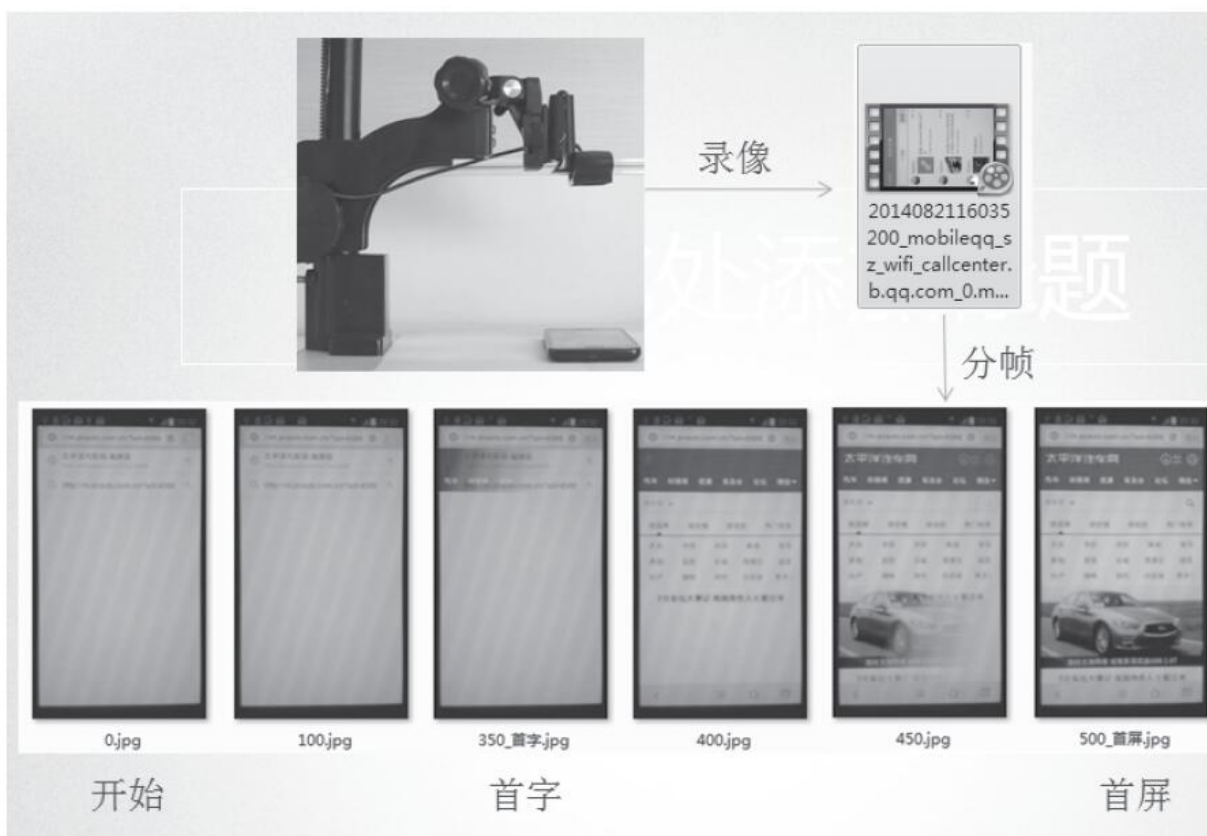


图7-15 录像方法示意图

7.3.3 整体方案

选定测试方法后，一起来看看浏览器网页打开速度的整体方案。

如图7-16所示，在手机侧，首先要做好手机环境准备，如清除缓存、打开被测场景所在的页面等。接着进行真正的测试动作，比如打开网页或者下载文件等。在进行测试动作的同时要打印日志并且抓取网络包。测试动作完成后通过adb命令将日志文件和网络包复制到PC端。最后需要将手机环境恢复，比如退出应用等。

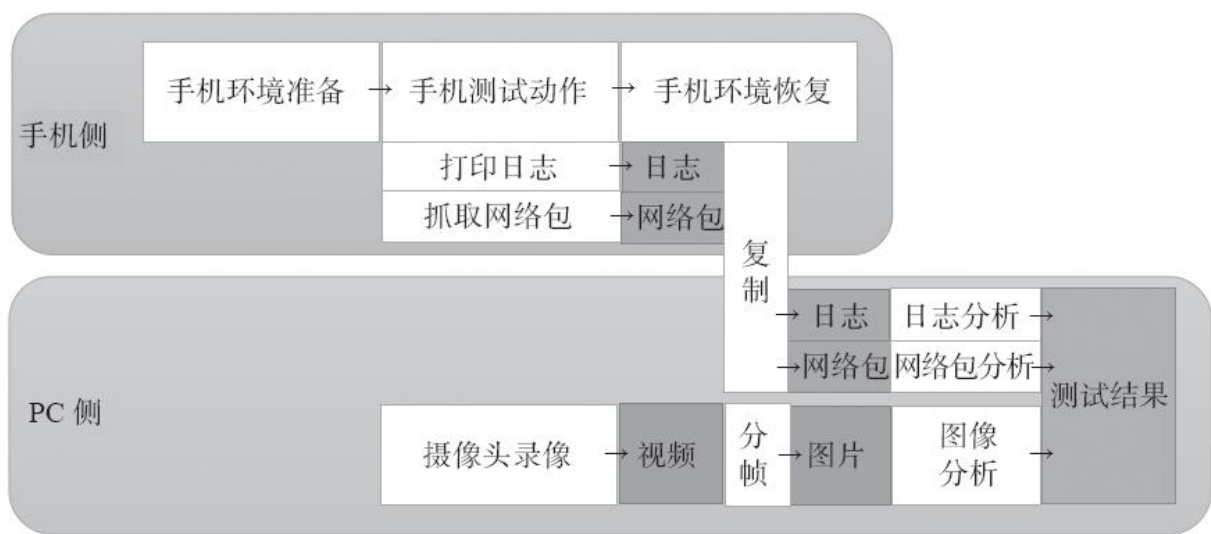


图7-16 浏览器网页打开速度的整体方案

在PC侧，需要对手机执行测试动作的全过程进行摄像头录像，并将视频分帧为图片。再加上之前从手机侧复制过来的日志文件和网络包，PC侧通过对应的分析算法就能分析出最终的测试结果了。

整体方案中的名词解释见表7-8。

表7-8 浏览器网页打开速度的整体方案中的名词解释

名词	解释	举例（以网页打开速度测试为例）
手机环境准备	执行手机测试动作之前的准备动作	打开浏览器，输入需要访问的网页 URL
手机测试动作	就是当前要测试速度的动作	点击“前往”按钮及等待网页打开这两个动作
打印日志	将手机测试动作过程的日志完整打印出来	保证在点击“前往”按钮前开始打印日志，在等待网页打开完成后才停止
抓取网络包	将手机测试动作过程的网络包完整抓取下来	保证在点击“前往”按钮前开始抓取网络包，在等待网页打开完成后才停止
摄像头录像	将手机测试动作过程完整录制下来	保证在点击“前往”按钮前开始录像，在等待网页打开完成后才停止
手机环境恢复	恢复手机环境以便进行下一次测试	删除缓存，关闭浏览器

7.3.4 解决关键问题

测试方法选定之后，还有很多关键问题需要解决，包括如何选择摄像头，如何将视频分帧，如何挑选关键帧，等等。除此之外，还要研究如何利用日志和网络包定位“慢在哪里”。

1.如何操作手机

手机浏览器网页打开速度测试的操作流程包括手机环境准备、手机测试动作和手机环境恢复。这些动作归纳起来就是点击、滑动、输入URL、等待、启动浏览器、关闭浏览器等。测试团队主要使用adb命令和sentevent方法来实现这些动作。

先介绍adb命令。adb是Android开发环境自带的工具。自动化测试需要的操作，使用adb命令基本都能完成，常用adb命令如代码清单7-1所示。

代码清单7-1 常用adb命令

```
adb install xxx.apk 安装
```

```
apk文件
```

```
adb shell am start -an com.xxx.xxx/.MainActivity 启动
```

```
App
```

```
adb shell am force-stop com.xxx.xxx 停止该
```

App
adb shell input keyevent KEYCODE_HOME 模拟

Android的

HOME按键

adb shell input text test_to_input 输入文字

adb shell input tap x y 模拟屏幕

touch操作

adb shell input swipe x1 y1 x2 y2 模拟屏幕滑动操作

adb devices 查看所有在线的

Android设备

adb -s deviceID shell input tap x y对指定

Android设备模拟屏幕

touch操作

大家经常用到的monkeyrunner其实只是将adb操作封装了一下而已。

接着介绍sentevent方法。sentevent方法说起来也是adb命令。单独把它列出来说的原因是，它具备adb shell input方法不具备的能力：模拟屏幕触碰事件。这个区别导致在设置了“显示触摸操作”之后，使用adb shell input方法点击屏幕不会产生白点，而使用sentevent方法则能产生白点。前文提到过，这个白点对于找准开始帧很重要！

sentevent方法的具体用法：

步骤1: 使用手机Shell下面的getevent命令获取手机事件。以按Power键为例, 命令执行结果为:

```
C:\Users\abc>adb shell
root@cancro:/ # getevent.....

/dev/input/event0: 0001 0074 00000001
/dev/input/event0: 0000 0000 00000000
/dev/input/event0: 0001 0074 00000000
/dev/input/event0: 0000 0000 00000000
```

以第一条为例分析一下获得的是什么。

·/dev/input/event0: 代表按键子系统, 包括Home、Menu、Back等按键。

·0001代表一个type。

·0074代表Power键的code (为16进制)。

·00000001代表value, 一般1代表按下, 0代表放开。

据查阅, device、type、code、value这四个参数正是sendevent命令需要的。



提示 具体关于getevent的参数可以使用getevent-h进行查看, 也可以参考Google提供的文档:

<http://source.android.com/devices/input/getevent.html>。

关于这些event的常量名的具体意义，可以参考Google的另一个很完整的文档：<http://source.android.com/devices/input/touch-devices.html>

。

步骤2：通过sendevent命令模拟手机事件。下面四条命令即可完成按Power键的操作，中间sleep的时间长度大于2s，系统就认为是长按，代码如下：

```
sendevent /dev/input/event0 1 116 1 (
```

```
0074转化为十进制后为
```

```
116)
```

```
sendevent /dev/input/event0 0 0 0  
sleep 3  
sendevent /dev/input/event0 1 116 0  
sendevent /dev/input/event0 0 0 0
```



注意

(1) 不同的手机的硬件中断触发事件略有区别，可以在adb下使用getevent进行测试。

(2) getevent得到的数值是16进制的，sendevent输入的参数是十进制的，注意进行转换。

(3) 如果依然没有效果，尝试先修改文件权限，执行su命令，再调用chmod 777/dev/input/event[x]。

使用RootTools库执行Linux层命令，不要使用Runtime.getRuntime()
() .exec () 。

sendevent方法就介绍这么多。实际使用中建议将sendevent方法进行封装后再使用，在TMQ官网可以下载笔者封装好的sendevent代码（以Nexus5手机为例编写）。

以上介绍的两种方法都是基于坐标的自动化测试方法。测试团队之所以没有用UIAutomation等基于控件的测试方法有以下三方面的原因：

- UIAutomation等基于控件的测试方法对于某些控件的识别能力不好。

- 基于控件的测试方法主要是解决机型适配问题。手机QQ浏览器速度测试使用的机型相对固定。

- 速度测试的操作相对简单，适配一个新机型工作量小。

2.摄像工具选择

摄像工具主要有两类：普通摄像头和手机摄像头。

而测试团队选择摄像头的标准是：成像质量高、帧率达标、容易控制、稳定性好。不同摄像头对比见表7-9。

综合考虑，测试团队选择了一款普通摄像头——罗技C920。

普通摄像头的可控性是它的软肋。PC上的驱动和摄像头客户端都能从罗技官网得到。但还有两个问题官网里没有答案，需要自己想办法解决：

表7-9 不同摄像头对比

	成像质量	帧率	可控性	稳定性
普通摄像头	拍摄手机屏幕，对摄像头的测光和对焦能力要求比较高。市面上 500 元以上的摄像头基本能满足要求	30 帧 / 秒	需要在 PC 端安装驱动以及摄像头客户端来控制摄像头。实现自动化比较麻烦	因为摄像头客户端对 PC 资源消耗较大，容易造成卡死。在部分机型上相对稳定
手机摄像头	拍摄手机屏幕，对摄像头的测光和对焦能力要求比较高。市面上高端手机基本能满足要求	30 帧 / 秒	使用 adb 命令控制手机摄像头开始和结束录像，操作方便。但需要区分控制录像手机和被测手机	因录像 App 稳定性而异

问题1：如何拍摄出高品质的录像

拍摄过手机的读者应该有经验，要拍出接近录屏效果的手机屏幕录像其实很难，必须把取景、对焦、测光、白平衡、去干扰等各个环节都处理好了才能得到高品质的录像。经过反复的探索，测试团队终于逐一找到了解决方案。

取景：使用翻拍架来固定手机和摄像头的相对位置，如图7-17所示。



图7-17 屏幕上显示触摸位置

对焦：取消自动对焦，并手动对焦到接近清晰，如图7-18所示。不要调到最清晰，最清晰容易产生衍射条纹。

测光：取消自动测光。手动调节合适的“曝光”和“增益”，如图7-19所示。

白平衡：保持自动，保证画面不会偏蓝或者偏红，如图7-19所示。



图7-18 触摸位置



图7-19 屏幕上显示触摸位置

去除干扰：用一个纸箱子盖住摄像头和手机，去除外部光照干扰。

经过以上步骤后，基本上可以得到一张接近录屏效果的录像，如图7-20所示。



图7-20 罗技C920录像分帧效果

问题2：如何用脚本控制开始录像和结束录像

因为罗技没有提供响应的接口可调用，所以我们选择了屏幕点击的方法，就是获取罗技摄像头软件Logitech Webcam Software的句柄，然后按相对坐标点击开始录制按钮，为此，测试组专门写了一个Exe程

序CameraController.exe（该程序收录于可下载的源文件中）。Exe程序调用方法为：

```
D:\abc>CameraController.exe start //启动  
  
Logitech Webcam Software  
D:\abc>CameraController.exe record //开始或结束录制
```

就此，开始录像和结束录像操作得以解决。

3.如何将视频分帧

视频分帧的工具有很多，测试团队选择的依据是：快速，能按原帧率分帧，分帧图片不失真，分帧图片尺寸小，可以实现自动化。

综合以上因素，测试团队最终选择了FFmpeg这个第三方工具。



注意 关于FFmpeg的介绍和工具下载可以访问以下两个网址：

工具介绍：<http://baike.baidu.com/item/ffmpeg>。

工具下载：<https://ffmpeg.org/>。

FFmpeg的使用方式：

ffmpeg-i videoFile-f image2-vf fps=fps=20 pngFiles

参数说明：（更多参数请使用ffmpeg-h命令查看）

`-f fmt` force format

`-i filename` input file name

`-vframes number` set the number of video frames to record

经过FFmpeg工具分帧，视频被转化为以毫秒时间命名的图片集。以每秒20帧为例，第一张图片命名为0000.jpg，第二张图片命名为0050.jpg，第三张图片命名为0100.jpg，依次类推。每张图片的名字代表的数值，就是这个图片与视频开始帧相隔的时间。这样，找到开始帧和结束帧就可以得知它们出现的时间了。

4.如何找准开始时间

前面介绍过，采用白点方法可以辅助找准开始时间。白点是通过系统设置→开发者选项→显示触摸操作来显示（仅Android手机支持）的。再次提醒：这个点击动作要用`sendEvent`方法实现，前文提过，用这种方法才能产生白点。效果如图7-21所示。左、右截图分别是点击前和点击后的效果。



图7-21 屏幕上显示触摸位置

在程序中，通过判断图片中的第一张白点出现的时间，即可以正确确定进入的时间。目前采用的算法是：判断图7-21中框框内白色值（RGB值的总和，框框需要用户预先定义）相对于前面一张的变化值，变化值达到一定的阈值，则判断为白点出现，目前方法较简单，但是识别率高，如代码清单7-2所示。

代码清单7-2 白点识别算法

```
public boolean findWhitePoint(AppUI des, Rect localArea) {
    boolean isChanged = false;
    // 获取第一张图片的

    WhiteSize, 表示白点出现前白点区域有多白

    if (0 == lastWhiteSize) {
        lastWhiteSize = des.getWhitePositionSize(localArea.x,
        localArea.y, localArea.width, localArea.height);
        return false;
    }
    // 获取第二张及以后的图片, 与前一张对比看白点区域有没有变白。如果变白了, 则表示白点出现

    else {
        int curSize = des.getWhitePositionSize(localArea.x, localArea.y,
        localArea.width, localArea.height);
        // 每个像素白色区域至少平均变化

        6. 才算是出现白色的点, 这里是一个经验阈值

        if (Math.abs(curSize - lastWhiteSize) > 6 * (localArea.width *
        localArea.height)) {
            isChanged = true;
        } else {
            isChanged = false;
            lastWhiteSize = curSize;
        }
    }
    return isChanged;
}
```

5.如何找准结束时间——首字时间

在检测到白点出现后, 紧接着将程序切换到找白屏的状态。白屏的饱和度为0, 当白屏后面的图片的饱和度大于0时, 代表首字出现, 如图7-22所示。左图尚未出现首字, 右图为出现首字的第一张图片。

这里引入了一个新的概念: 饱和度。

因为前文介绍的图像对比算法找结束帧有一个明显的短板: 它必须先指定一个标准图片。而对于首字时间这个指标来说, 标准图片是

不确定的，因为用户不能确定一个网页是会先显示文字还是先显示图片。因此，这里需要一种新的算法来找标准图片不确定的结束帧。饱和度算法由此提出。

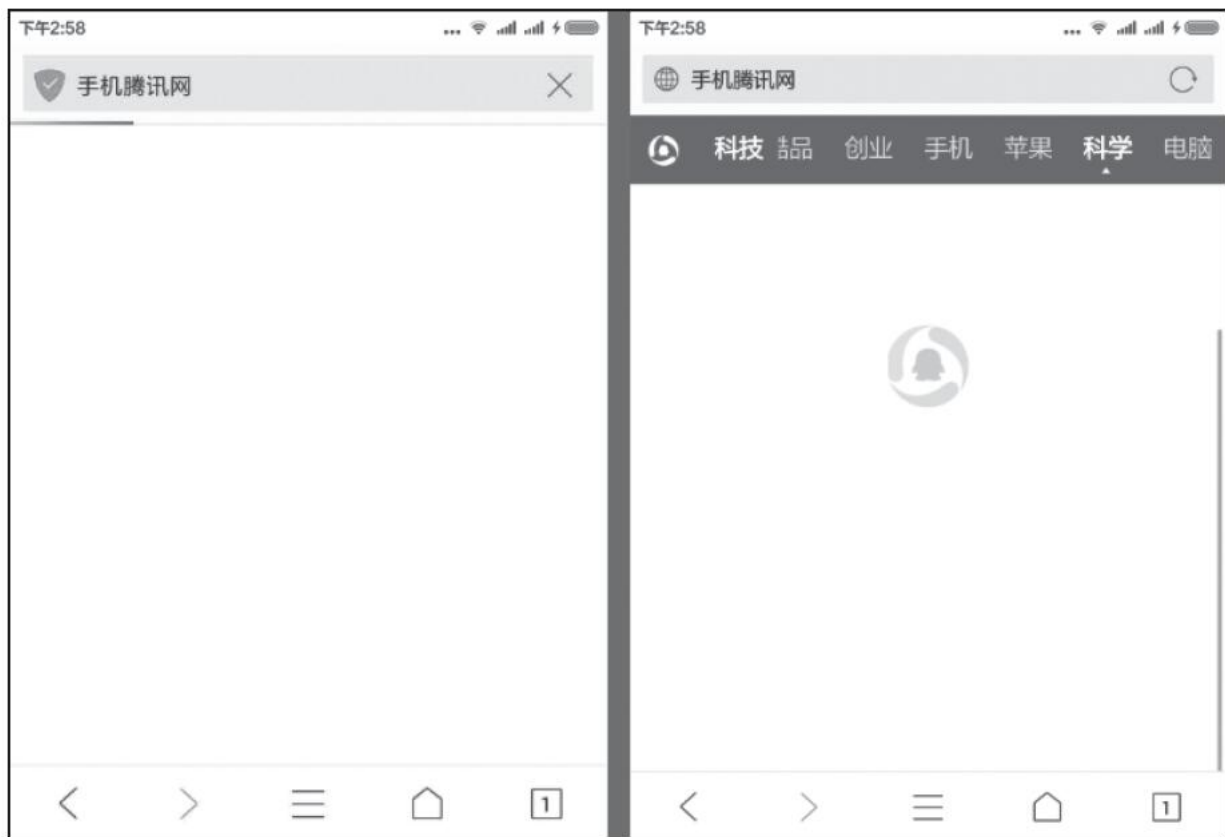


图7-22 首字图片

图片饱和度算法： 将分帧图片页面加载区域分成5×8（例子）个格子，记为total=40，统计不是白色的格子（颜色的复杂度大于某个阈值）的格子个数，记为count，饱和度=count/total，如图7-23所示。

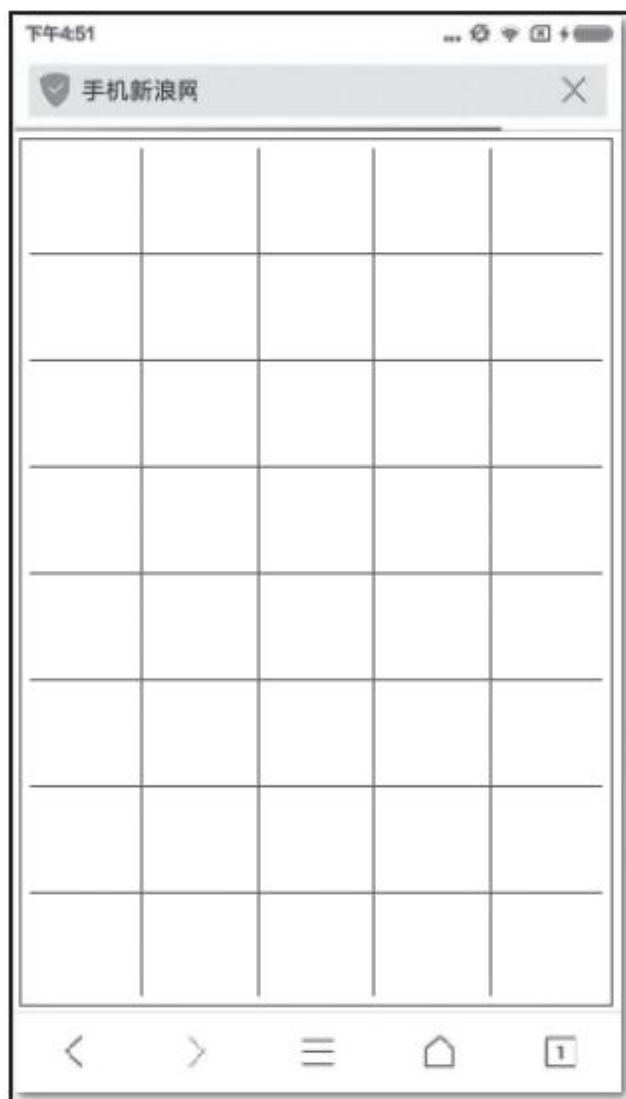


图7-23 图片饱和度算法

代码如代码清单7-3所示。从这个算法的说明中可以看出，不管页面什么地方显示出内容，也不管这些内容是文字还是图片，它都能第一时间识别出来。

代码清单7-3 获取图片饱和度算法

```

public boolean[][] getComplexityStatus(Rect loadingArea) {
    boolean[][] res = null;
    if (loadingArea != null) {
        //指定需要分析区域的范围

        int x = loadingArea.x;
        int y = loadingArea.y;
        int width = loadingArea.width;
        int height = loadingArea.height;
        //指定什么叫白色（因为录像出来的白色总是有点灰）

        Color CurWhite = getWhite(loadingArea);
        // 70像素一个格子，看屏幕有多少个格子

        int row = height / PerformanceAnalyzer.getStepHeight();
        int column = width / PerformanceAnalyzer.getStepWidth();
        res = new boolean[row][column];
        // 将需要分析的区域分成多份

        Rect[][] sub = new Rect[row][column];
        int subW = width / column;
        int subH = height / row;
        for (int i = 0; i < column; i++) {
            for (int j = 0; j < row; j++) {
                int xx = x + subW * i;
                int yy = y + subH * j;
                sub[j][i] = new Rect(xx, yy, subW, subH);
            }
        }
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < column; j++) {
                if (!res[i][j]) {
                    int xx = sub[i][j].x;
                    int yy = sub[i][j].y;
                    int ww = sub[i][j].width;
                    int hh = sub[i][j].height;
                    float p = 0.001f;
                    //计算当前格子非白色的像素点的比例

                    p = this.getColorPercentage(CurWhite, xx, yy, hh, ww, 140, 0);
                    //如果当前格子非白色的像素点超过

                    5%, 则认为这个格子已经显示内容

                    if (p > 0.05f) {
                        res[i][j] = true;
                    }
                }
            }
        }
        return res;
    }
}

```

6.如何找准结束时间——首屏时间

对于网页打开速度来说，结束时间就是首屏出现的时间。对于普通的网页，直接用图像对比的方法就能找到首屏。

以图7-24为例，选取视频分帧的最后一帧为标准图片（首屏标准）。从前往后将每张图片与标准图片对比，当出现与标准图片一致的图片即为首屏（图7-24中的6图）。

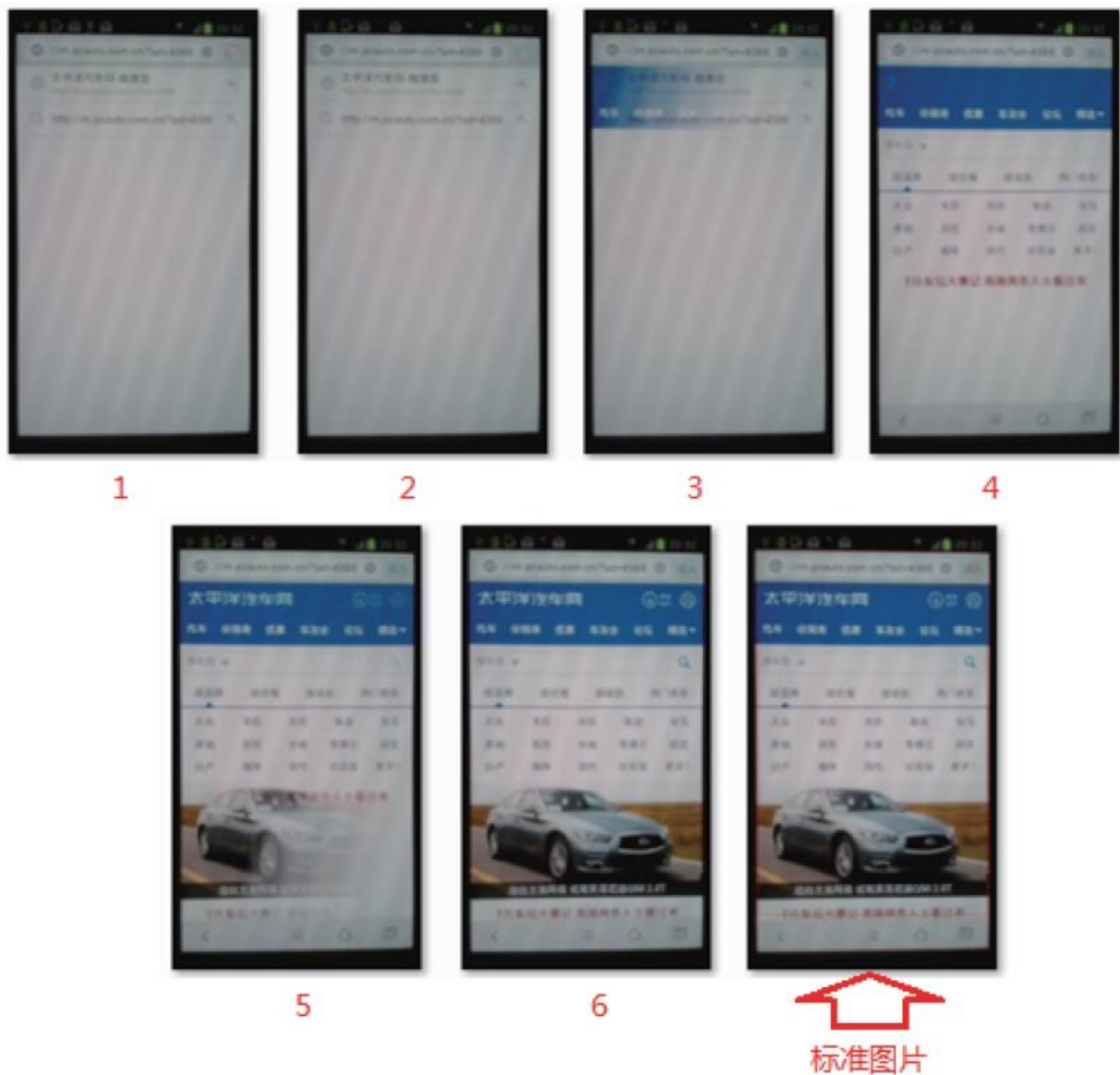


图7-24 通过标准图片找首屏

图像对比算法或者工具有很多，测试团队选择的标准是：准确、快速、可设置灵敏度。最终，测试团队选定了感知哈希算法。

感知哈希（hash）算法描述了一个有可比较的哈希函数的类。图像特征被用于生成独特的指纹，这个指纹相当于这张图片的特征参数，

而且这个指纹是可比较的。如果hash值是不同的，则数据（图片内容）也是不同的；如果hash值是相同的，则数据也是相似的。（因为可能存在hash冲突，相同的hash值会产生不同的数据）感知哈希算法是目前图片搜索领域的常见算法之一。感知哈希特征参数的提取原理如图7-25所示。

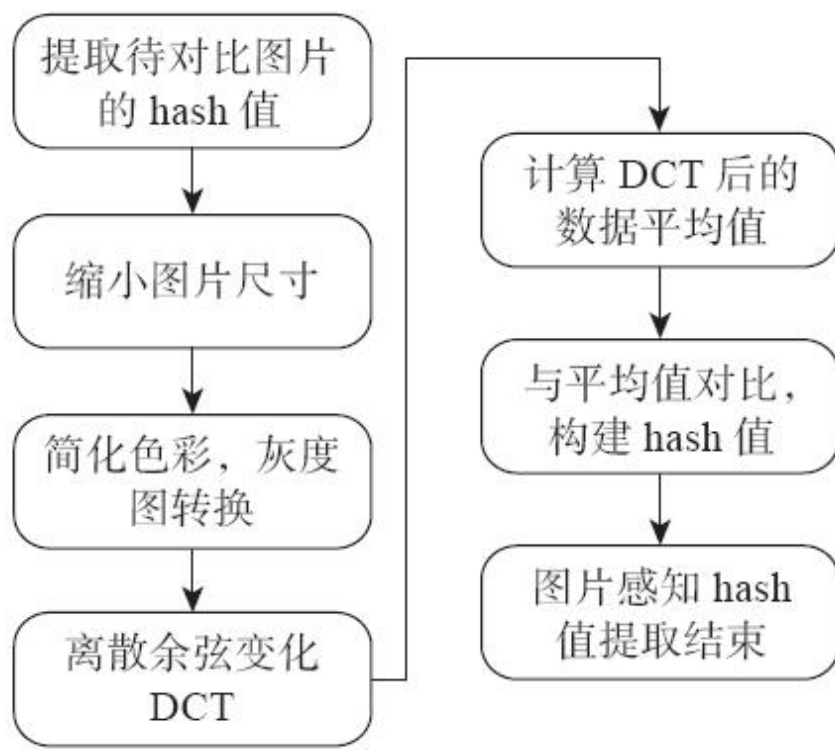


图7-25 感知哈希特征参数的提取原理

识别过程比较简单，通过对比待识别图片和标准图片特征参数的相似度（汉明距离）来判断图片是否相同，识别流程如图7-26所示。

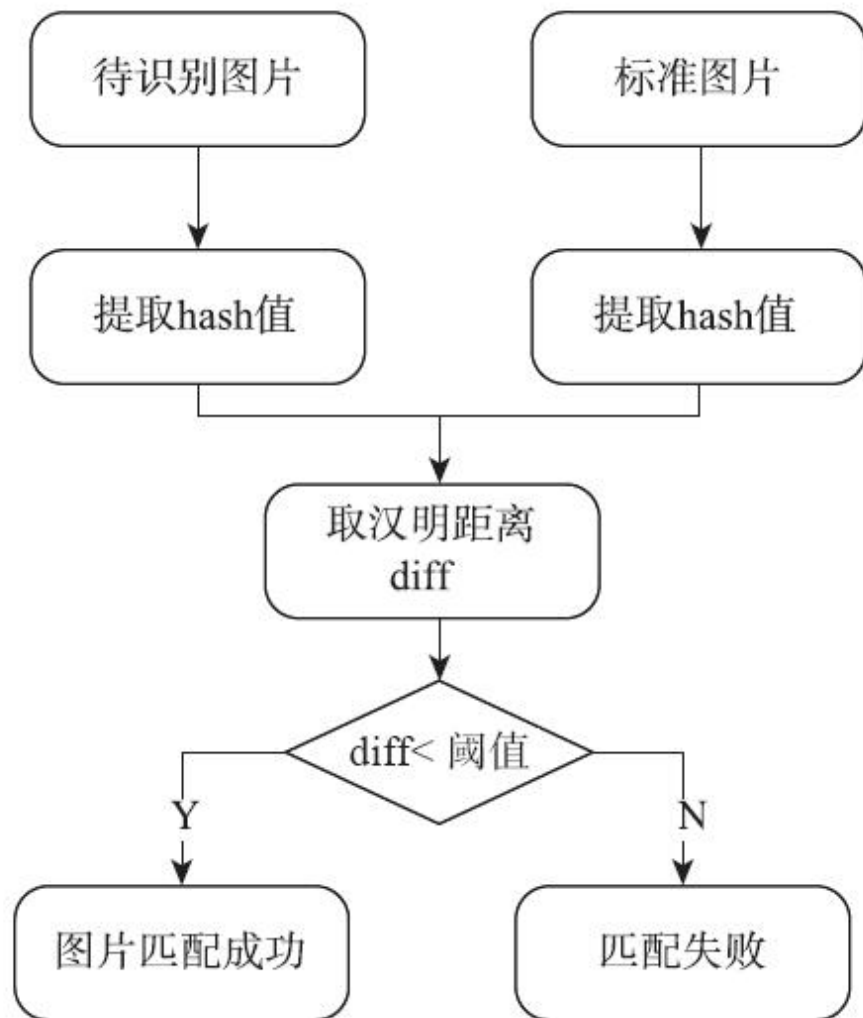


图7-26 图片搜索识别流程

到这里，通过图像分析方法，已经得到测试团队最初选定的两个关键指标的度量值——首字时间和首屏时间。接下来，看如何通过打印日志和网络包分析得出慢在哪里。

7.当网页带有幻灯片时如何找准首屏

对于另外一些网页，打开后会有大图幻灯片滚动，如图7-27所示，这种情况给首屏的找准带来了新的困难。此时录像的最后一帧播放的幻灯片是不确定的，就不能直接用它作为标准图片了。

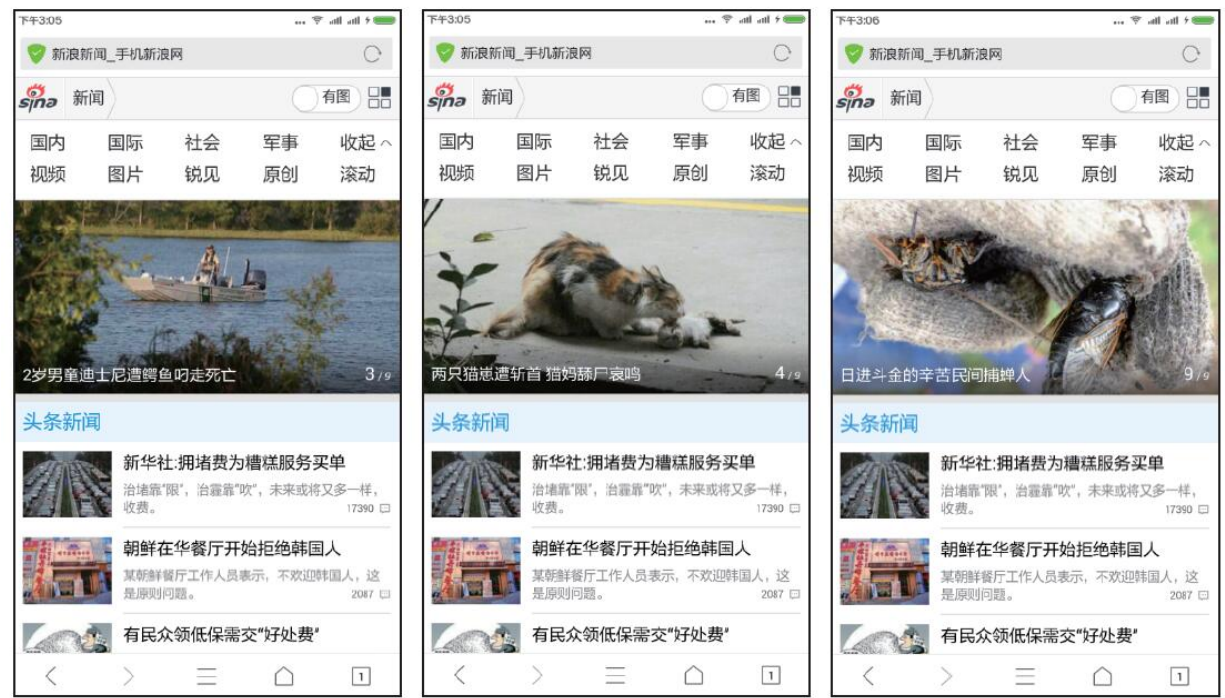


图7-27 有大图幻灯片滚动的网页

于是需要引入一种新的算法：标准图片搜索算法。

该算法的逻辑如下：从最后一张图片开始往前搜索不同的图片，分别记录为标准图片1、标准图片2.....标准图片N。当发现图片开始循环时，搜索算法结束，于是就产生了一个标准图片集合。接下来的首屏找准方法跟普通网页一样，只是标准图片从原来的一张变成了多张。只要当前图片和标准图片中的任意一张相同，即被识别为首屏。

除了大图幻灯片滚动这种情况外，还有很多各种各样的情况会给首屏找准带来困难，比如广告、自动播放的视频。所以首屏算法还需要留最后一手：人工指定首屏图片。这种方法需要人工参与，但是灵活性比较好，适合应对其他算法都无法解决的特殊情况。

8.打印日志计时法如何分析“慢在哪里”

通过打印日志，可以得知程序打开网页的每个过程的耗时。业界也有TraceView、SystemTrace、Oprofile等性能优化工具。但对于浏览器这种特殊形态的程序，有一个更方便的日志形式来获取和查看这些信息——chrome trace文件。分析网络模块的性能都可以用到这个日志格式。

chrome trace文件是chrome浏览器自带的用于打印浏览器每个线程的工作状态的文件格式。打印的内容需要开发人员自己添加，查看方式则是通过chrome浏览器访问chrome: //tracing/查看，如图7-28所示。

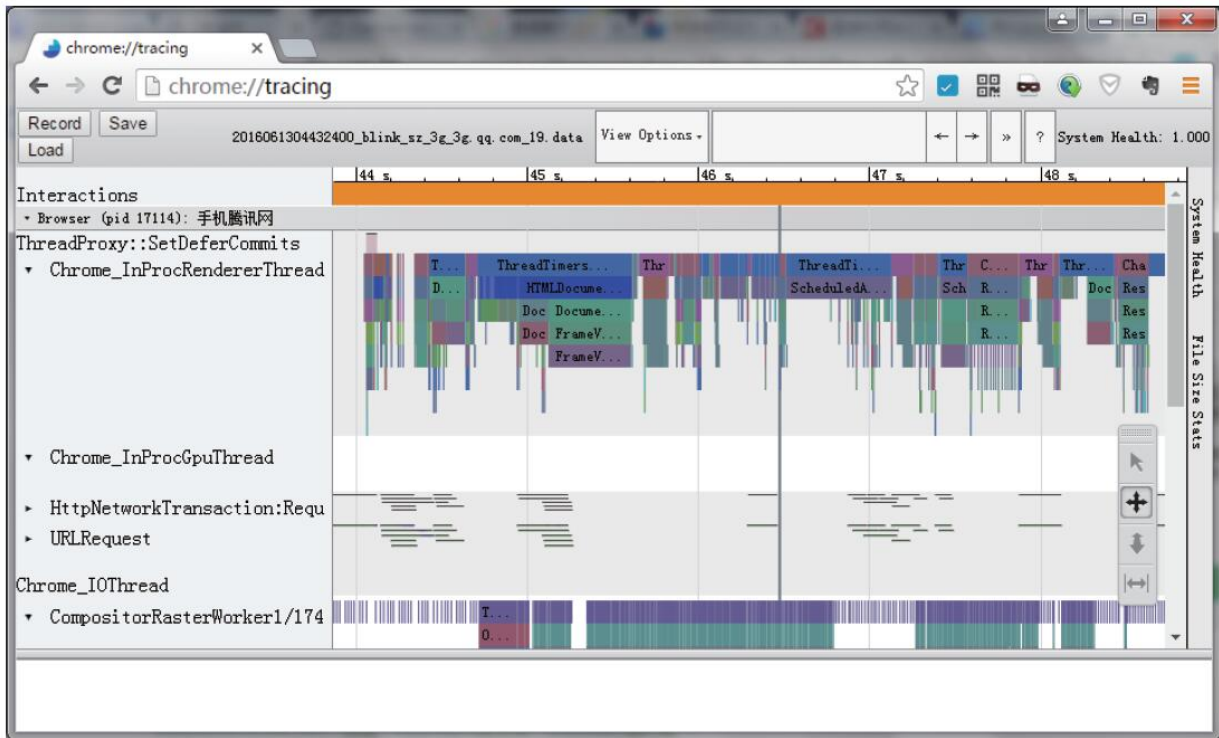


图7-28 trace文件展示图



注意 trace文件介绍:

<https://www.chromium.org/developers/how-tos/trace-event-profiling-tool>

如何抓取trace文件: <https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/recording-tracing-runs>



注意 因为trace不是本节介绍的重点，所以这里点到为止，不再展开介绍。详情可参考chromium官网www.chromium.org。

9.网络包分析法如何分析“慢在哪里”

从浏览器网页打开速度的整体方案图7-15中可以看到，打开网页的时候会全程抓取网络包，然后将网络包复制到PC进行分析。前文也已经提到过，pcap包中包含的信息很多，需要提炼出一些关键的指标并将它自动化分析出来。这样可以通过这些指标快速判断一次测试的网络包有没有问题。如果有问题，再打开网络包查看详细数据。最终，测试团队确定的关键网络指标是：流量、网络完成时间和主资源耗时。

举一个例子说明网络包是怎么辅助定位问题的。有一次笔者测试赶集网的一个页面，发现手机QQ浏览器的速度比系统浏览器慢很多。检查流量和网络完成时间发现都超过了系统浏览器。打开网络包查看，发现手机QQ浏览器有两个50K以上的大资源重复请求了，如图7-29粗框处所示，而系统浏览器则不会重复请求。

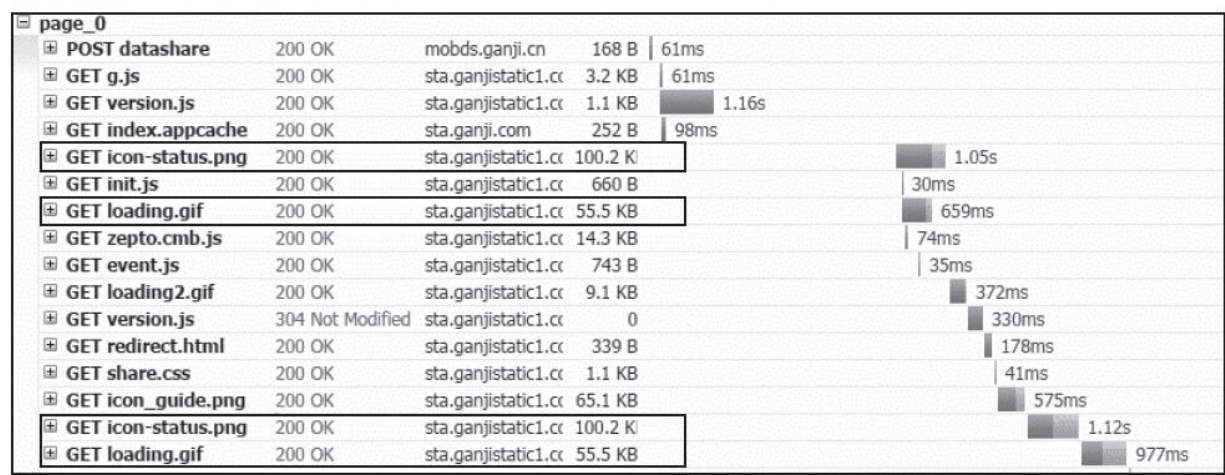


图7-29 手机QQ浏览器重复请求了两个大资源

开发人员进一步定位问题，发现该问题是由手机QQ浏览器的云加速代理服务器在应答含有If-modified-since包头字段的Get请求时处理不

当导致的。

关于网络分析还有一个小技巧，就是可以在点击go打开网页的同时发送一个ping网络包，这样就可以轻松知道在网络包里是什么时间点击go的了。这对于分析主资源请求的延时很有帮助！如图7-30所示，ping包发出时间是39.405267s，主资源请求发出时间是39.610197s。两者相差205ms，属于正常范围。如果这个延时变长，就需要分析是哪里出现了问题。

No.	Time	Source	Destination	Protocol	Info
131	39.405267	10.168.204.242	9.9.9.9	ICMP	Echo (ping) request id=0x0048, seq=2/512, ttl=64
134	39.580907	10.168.204.242	163.177.73.12	SPDY	60517 > http-alt [PSH, ACK] Seq=1 Ack=1 win=13600 Len=40
135	39.581066	10.168.204.242	163.177.73.12	SPDY	SETTINGS MAX_CONCURRENT_STREAMS=1000 INITIAL_WINDOW_SIZE=10485760
136	39.610197	10.168.204.242	163.177.73.12	SPDY	SYN_STREAM Stream=1 Request="GET http://i.ifeng.com/ HTTP/1.1"
145	39.764148	163.177.73.12	10.168.204.242	SPDY	SYN_REPLY Stream=1 Response="200 HTTP/1.1"

图7-30 利用ping包计算主资源延时

10.对测试结果进行数据处理

因为速度测试存在网络波动等影响因素，测试结果会有误差。为了将误差控制在可以接受的范围内，对测试的数据需要做以下处理：

- (1) 测试足够的次数。
- (2) 去除异常数据。
- (3) 取得平均值。

如何让测量值尽量接近被测值是一门很高深的学问。在这方面我们做的研究和尝试比较有限，这里只分享个人觉得最有参考意义的一些发现：

（1）网页打开速度的数据点并非如很多人想象的那样成正态分布，而是成对数正态分布。

（2）数据处理算法的科学严谨性和可操作性之间需要取得一个平衡。

关于（2），我们有过失败教训。最初我们用正态分布的模型求测试数据的期望值，后来发现这种方式算法太复杂，也不容易发现和定位问题，就改用了切尾均值（指在一个数列中，去掉两端的极端值后所计算的算术平均数）。

11.如何展示测试结果

为了方便查看结果，开始时间、结束时间以及网络包分析等测试结果保存在本地的同时会上传到服务器展示。原始数据经过一定的数据处理剔除异常数据后取平均值生成最终结果，如图7-31和图7-32所示。

浏览器	代理	循环次数	首字时间	首屏时间	完成时间	网络时间	流量数据
blink	direct	1	3650.0	4700.0	4700.0	4538.0	713.399
blink	direct	2	1666.0	4263.0	4263.0	1581.0	671.854
blink	direct	3	1600.0	2050.0	2050.0	1620.0	677.556
blink	direct	7	1617.0	3087.0	3087.0	1641.0	673.526
blink	direct	8	1700.0	2750.0	2750.0	1743.0	672.632
blink	direct	9	1568.0	1568.0	1568.0	1629.0	672.778
blink	direct	10	1600.0	2500.0	2500.0	1678.0	671.325

图7-31 详细数据

测试网址	接入点	地区	指标	blink代理
3g.qq.com	Wi-Fi	sz	首字时间	1135
			首屏时间	1215
			完成时间	1215
			网络时间	2268
			流量数据	291

图7-32 平均值汇总数据

7.3.5 速度优化效果

手机QQ浏览器网页打开速度测试这套技术方案，测试团队根据不同的目的制定了四种不同的测试类型，见表7-10。

表7-10 网页打开速度测试四种不同的测试类型

测试类型	测试目的	测试周期	测试效果
Dailybuild 版本测试	及时发现每天构建版本有没有性能倒退	1 天	监控 2 年，发现 5 次明显性能倒退，20 次小幅性能倒退
上线前性能测试	衡量版本网页打开速度是否达到发布标准	1 个版本	测试 2 年共 20 个版本，2 次发现版本不符合发布标准
Top 站点测试	发现手机 QQ 浏览器相对竞品慢的站点，以便针对性优化	1 个月	每月发现 3 ~ 5 个相对竞品落后的站点
优化版本测试	对比新旧版本，验证开发的速度优化代码提交后是否有效	不定时	帮助开发验证 30 个优化点的有效性

- 脚本运行效率：一部手机每天晚上可以测试三个站点与竞品对比（每个站点测试20次）。
- 人力投入：每次测试（一部手机测试三个站点与竞品对比）需要投入0.5~1天的人力，包括测试前的准备和测试后的校验审查。

7.4 手机QQ浏览器多窗口按钮速度实践案例

上一节讲解了手机QQ浏览器网页打开速度测试，本节以手机QQ浏览器功能类多窗口按钮速度测试为例，详细讲解功能类速度测试如何开展。

7.4.1 为什么要做多窗口按钮速度测试

多窗口按钮是实现浏览页面之间相互切换的功能按钮，浏览功能是用户的核心诉求，手机QQ浏览器提供多个页面浏览功能，使我们可以打开多个窗口，每个窗口浏览一个页面，看完一个页面，切换到另一个页面继续浏览，提高浏览的顺畅性。多窗口按钮是通往页面切换的必经路径，从数据上看，用户使用多窗口实现切换页面浏览非常频繁，故我们对这类用户使用频繁的按钮进行相应的速度测试。另外从用户的习惯和数据统计看，用户使用三个窗口的情况比较多，故我们在挑选窗口数时，选择了三个窗口。

7.4.2 什么是多窗口按钮速度测试

图7-33中左边图为浏览三个页面图，其中框线中为多窗口按钮，右边图为点击多窗口按钮后多窗口界面图，而我们要测试的多窗口按钮速度，指的就是从点击多窗口按钮到右边界面图显示出来的时间。



图7-33 多窗口按钮以及点击后效果图

在这里我们可思考一下，为什么是速度测试，而不是内存或者流畅度等测试呢？我们是这么思考的，用户点击多窗口按钮，目的是什么呢？当然不是打开多窗口界面，而是切换浏览页面，那么用户在这个位置最期望什么呢？是以最快的速度切换到自己想要浏览的页面，故该位置速度就成为非常重要的指标，所以我们对浏览器多窗口按钮进行了速度测试。

7.4.3 多窗口按钮速度测试影响因素和测试方法

做多窗口按钮速度测试之前，我们需要分析一下，对于多窗口按钮速度测试来讲，有哪些因素会影响到多窗口按钮速度测试，这里列举一下会产生影响的因素：

·**手机：**不管是自身版本对比还是跟竞品对比，建议为同一部手机和同样的环境，以保证对比的可靠性。如果不是同一部手机，手机性能不同，则导致无可比性。

·**网络：**我们是三个窗口（最常见的用户打开的窗口数），三个窗口是网页，必须确保网络已经拉取完成。因为如果有网络未拉取完成，那么在点击多窗口按钮时，就可能存在又拉取网页、又打开多窗口按钮的现象，事件并发执行，对资源的消耗是不一致的，故我们在测试三个多窗口时，都是保证网页拉取完成，同时在自身版本做对比或者跟竞品做对比时，都需要保证网页一致。

·**运行的程序：**保证其他App也是一致的情况，手机的资源是一定的。如果有其他App在耗用资源，那么被测对象就可能受到影响。这里有个真实的案例，之前我们在测试时，有其他App一直在后台运行，导致我们多轮数据非常不稳定，最后查找原因为其他App的影响。

·**样本**：为了保证得到的数据是正确的，建议测试多轮，这样不仅可以看到多轮数据是否稳定，排除其他影响，而且也能看到多轮趋势。相对来讲，采用样本越多，越能得出精准数据。

当我们排除影响因素后，就可以开始做多窗口按钮速度测试了，首先对于之前介绍的速度测试方法，我们来看看选择什么样的测试方法，能快速、高效地获取准确数据。因我们做的是多窗口按钮速度测试，之前提到了几种测试方法，我们逐一来衡量，看哪种更适合多窗口按钮速度测试。

·**掐表计时法**：新窗口打开速度低于1s，采用该方法准确度无法保障，故未采用。

·**打印日志计时法**：该方法对于自身产品是可用的，但是如果要跟竞品对比，就无法达到，而多窗口打开速度需要和竞品对比，故也不采用这种方法。

·**图像分析计时法**：该方法能满足需求，但只是一个具体的时间，而对于多窗口打开速度而言，我们不仅仅希望得到总体数据，还希望得到详细数据，如哪里慢了，是什么原因导致的慢，故该方法也未采用。

·**Hook方案计时法**：本次多窗口打开速度我们选用这种方法，因这种方法能知道每个View的时间，以此来找到导致慢的原因。

综上所述，在多窗口按钮速度测试案例上我们采用了**Hook**方案计时法。

7.4.4 如何进行多窗口按钮速度测试

对于多窗口按钮速度测试，我们先思考一下整个过程，然后从整个过程中，看看我们可以获得哪些数据。点击多窗口按钮，弹出多窗口界面，这个是直观的感受，也是我们需要获得的时间，但是整个过程具体在做什么呢？从技术角度来讲，首先是多窗口按钮被点击，触发相应事件，然后进行多窗口界面View等的绘制，当多窗口界面所有View绘制完成后，多窗口界面就显示出来。从以上分析可知，从点击多窗口按钮到所有View的绘制完成时间就是多窗口按钮的打开速度，这里View的绘制快慢决定了打开速度的快慢，我们通过hook（请参考7.2.4hook方案计时法）方法，在浏览器中打开三个窗口，等待每个窗口中的页面拉取完成，然后点击多窗口按钮直到所有View绘制完成为止，整个操作对应的信息会输出到Logcat日志中，获取相应View的绘制时间详情如下：

1.获取总时间以及各个View绘制时间（View的onDraw的时间总和）

通过hook方案中的Logcat日志，找到总时间，通过查找MethodHook关键字，得到所有关于该关键字的信息如图7-34所示，其中最后一行“[click]null cost: 689”就是多窗口打开总时间。

```

Line 73: D/MethodHook(11261): frame[17367400] draw [0,60][1080,1920] cost 31
Line 75: D/MethodHook(11261): frame[17367401] draw [0,1776][1080,1920] cost 44
Line 77: D/MethodHook(11261): frame[17367402] draw [0,1776][1080,1920] cost 56
Line 79: D/MethodHook(11261): frame[17367403] draw [0,1776][1080,1920] cost 77
Line 81: D/MethodHook(11261): frame[17367404] draw [0,60][1080,1920] cost 100
Line 83: D/MethodHook(11261): frame[17367405] draw [0,1776][1080,1920] cost 111
Line 85: D/MethodHook(11261): frame[17367406] draw [0,1776][1080,1920] cost 123
Line 221: D/MethodHook(11261): frame[17367407] draw [0,0][1080,1920] cost 387
Line 275: D/MethodHook(11261): frame[17367408] draw [0,0][1080,1920] cost 397
Line 277: D/MethodHook(11261): frame[17367412] draw [0,0][1080,1920] cost 418
Line 279: D/MethodHook(11261): frame[17367413] draw [0,0][1080,1920] cost 424
Line 283: D/MethodHook(11261): frame[17367414] draw [0,0][1080,1920] cost 439
Line 285: D/MethodHook(11261): frame[17367415] draw [0,0][1080,1920] cost 455
Line 287: D/MethodHook(11261): frame[17367416] draw [0,0][1080,1920] cost 473
Line 353: D/MethodHook(11261): frame[17367417] draw [0,0][1080,1920] cost 493
Line 357: D/MethodHook(11261): frame[17367418] draw [0,0][1080,1920] cost 511
Line 359: D/MethodHook(11261): frame[17367419] draw [0,0][1080,1920] cost 531
Line 363: D/MethodHook(11261): frame[17367420] draw [0,0][1080,1920] cost 550
Line 365: D/MethodHook(11261): frame[17367421] draw [0,0][1080,1920] cost 566
Line 367: D/MethodHook(11261): frame[17367422] draw [0,0][1080,1920] cost 577
Line 411: D/MethodHook(11261): frame[17367423] draw [25,695][1055,1920] cost 600
Line 417: D/MethodHook(11261): frame[17367424] draw [0,0][1080,1920] cost 616
Line 419: D/MethodHook(11261): frame[17367426] draw [0,0][1080,1920] cost 639
Line 421: D/MethodHook(11261): frame[17367427] draw [0,0][1080,1920] cost 667
Line 423: D/MethodHook(11261): frame[17367428] draw [0,236][1080,1920] cost 673
Line 425: D/MethodHook(11261): frame[17367429] draw [84,768][126,809] cost 689
Line 1611: D/MethodHook(11261): [click] null cost: 689

```

图7-34 多窗口打开总时间

为了找到影响速度最大的View，我们又通过hook方案中的Logcat日志，获得单个View时间，可通过这样的方式获取：如获取frame17367429的时间，通过查找17367429，获得的信息如图7-35所示，其中第三行“cost: 5”就是指该View的时间为5ms。

```

Line 381: D/HookDebug(23994): [frame][17367429] start----- at 1456104832479
Line 383: D/HookDebug(23994): [frame][17367429] draw: Rect(84, 768 - 126, 809)
Line 387: D/HookDebug(23994): [frame][17367429] end----- cost: 5
Line 425: D/MethodHook(11261): frame[17367429] draw [84,768][126,809] cost 689

```

图7-35 View17367429耗时图

为了能比较清晰地知道View耗时对比，我们将每个View耗时图形化，图7-36所示为两个版本的View耗时对比图，从图中比较容易看出View耗时对比，也容易找到耗时多的View，以供开发人员优化。

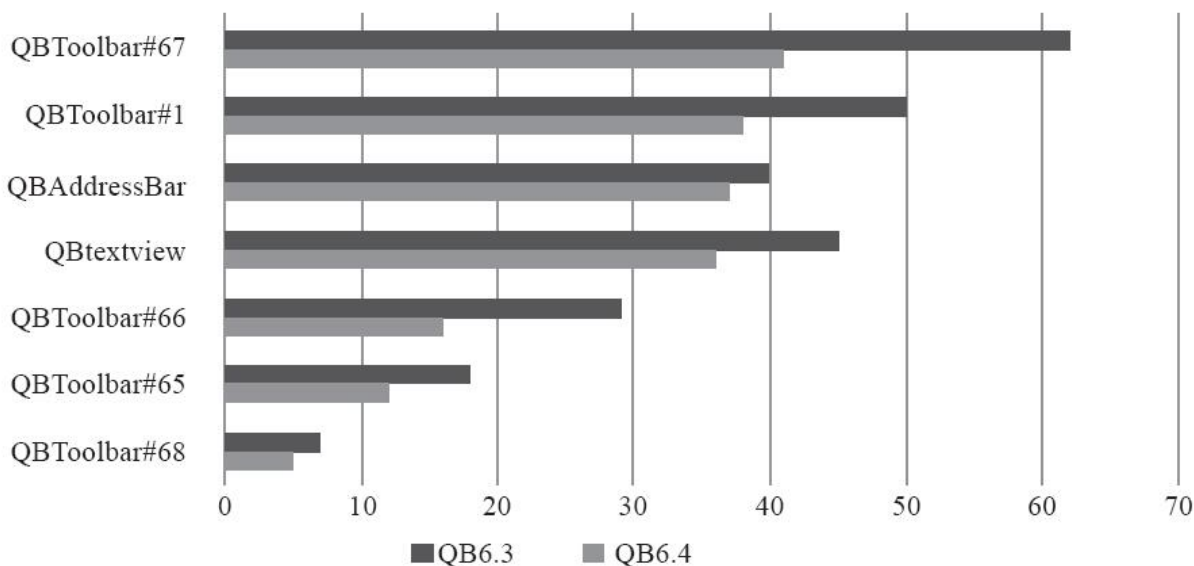


图7-36 QB6.4 VS QB6.3多窗口View绘制时间（ms）

获得各个View的时间后，我们可找出哪个View耗时最多，哪个View应该优化。比如图7-36中，可以看到耗时最多的View是QBToolbar#67，进而在6.4版本中针对耗时多的View进行优化。另外从View绘制机制中了解到，绘制时间还会跟过度绘制倍数有一定的关系，下面我们看看如何获得UI过度绘制（UI过度绘制简单说就是指在一个界面中有很多元素，当只需要更新某一小块的元素时，App却把所有的元素都刷新一遍，这就会造成过度绘制）。

2. 获得UI过度绘制倍数

通过Hook方案中的Logcat日志，查找overdraw来获得UI过度绘制倍数overdraw: 1.2942646，这样代表过度绘制倍数为1.2942646。过度绘制倍数越高，绘制速度越慢，故可通过减少过度绘制倍数来提升速度。以浏览器两个版本为例，之前过度绘制为2.69，优化后，过度绘制为1.29。View绘制时间和是否存在过度绘制都是影响速度的重要因素，除此之外，UI布局也非常重要，下面我们再来看看布局是否合理。

3.UI布局是否合理

通过Hierarchy Viewer（使用方法可参考网络）观察UI Tree的深度和View总数，来衡量UI布局是否合理。

在旧版本中，我们通过观察多窗口Hierarchy树形结构，打开新浪、腾讯、搜狐三个窗口，发现View 22个，Tree深度为6级，而深度对于渲染是非常关键的点，深度越深，渲染越慢，故提出需要优化Tree的深度。经过开发优化，新的版本中同样的情况多窗口有26 View，Tree深度为5级，然而渲染的总时间减少了，优化有了一定的效果。上面是从绘制角度和布局角度来深度分析多窗口按钮速度耗时点，下面我们可以再深入一些，做到函数时间。

4.函数Trace时间

通过记录Trace（使用方法可参考网络），查看排名TOP的函数，为开发人员提供重点优化的关键点，如图7-37所示。

Name	Incl Cpu Time %
27 com.tencent/mtt/browser/multiwindow/view/MultiWindowView.onMeasure (II)V	14.3%
28 android/view/ViewGroup.dispatchDraw (Landroid/graphics/Canvas;)V	14.2%
29 android/view/ViewGroup.drawChild (Landroid/graphics/Canvas;Landroid/view/View;)Z	14.2%
30 android/view/View.draw (Landroid/graphics/Canvas;Landroid/view/ViewGroup;)Z	14.2%
31 com.tencent/mtt/browser/multiwindow/view/WindowItemContainer.onMeasure (II)V	13.6%
32 android/view/HardwareRenderer\$GLESRenderer.buildDisplayList (Landroid/view/View;Landroid/graphics/Canvas;)Z	13.5%
33 android/view/View.getDisplayList ()Landroid/view/DisplayList;	13.5%
34 android/view/View.getDisplayList (Landroid/view/DisplayList;Z)Landroid/view/DisplayList;	13.4%
35 com.tencent/mtt/browser/multiwindow/view/WindowItemContainer.synchronizeStackViews	10.6%

图7-37 Trace图

总结：通过四个方面比较详细的数据分析，经过优化，多窗口在新版本中的打开时间从原来的864ms减少到468ms，缩短45.8%。如果想让测试对于关键点能起到正向引导开发优化的良好作用，测试应该不仅仅提供数据，还应该对于总体数据进行详细的分析，提供比较全面的分析来正向指导开发优化，以此来提升产品口碑和测试口碑。

7.5 本章小结

本章介绍了速度测试常用的几种测试方法，并以两个案例详细讲述了图像分析计时法和Hook方案计时法在手机浏览器项目中的实践过程及效果。测试方法的选择应该因地制宜，不应该一味追求高级技术方案。但无论使用什么测试方法，速度测试的开展都应该按照“测试场景选择→测试方法选择→测试方案实现”的步骤依次进行。通过对指定场景的速度度量和发现“慢在哪里”，最终帮助研发团队优化产品性能。

第8章 视频性能测试案例

本章通过浏览器视频性能测试案例，详细讲解和剖析视频播放首帧响应时间测试方案以及实现原理，它不是其他章节所述的框架的直接应用，而是自动化测试的工具改造及应用的代表案例。读者阅读本章需要一定的编程基础，主要包括Android SDK和NDK基础编程、OPENCV图形识别和相似度对比技术、FFMPEG视频解码技术以及Java和Javascript之间通信的相关知识。同时，本章的工具开发涉及代码也比较多，建议读者在学习本章内容时，下载相应的代码多加实践，以理解其中的精华。为使读者更好地阅读本章，我们整理了本章知识点，如图8-1所示。

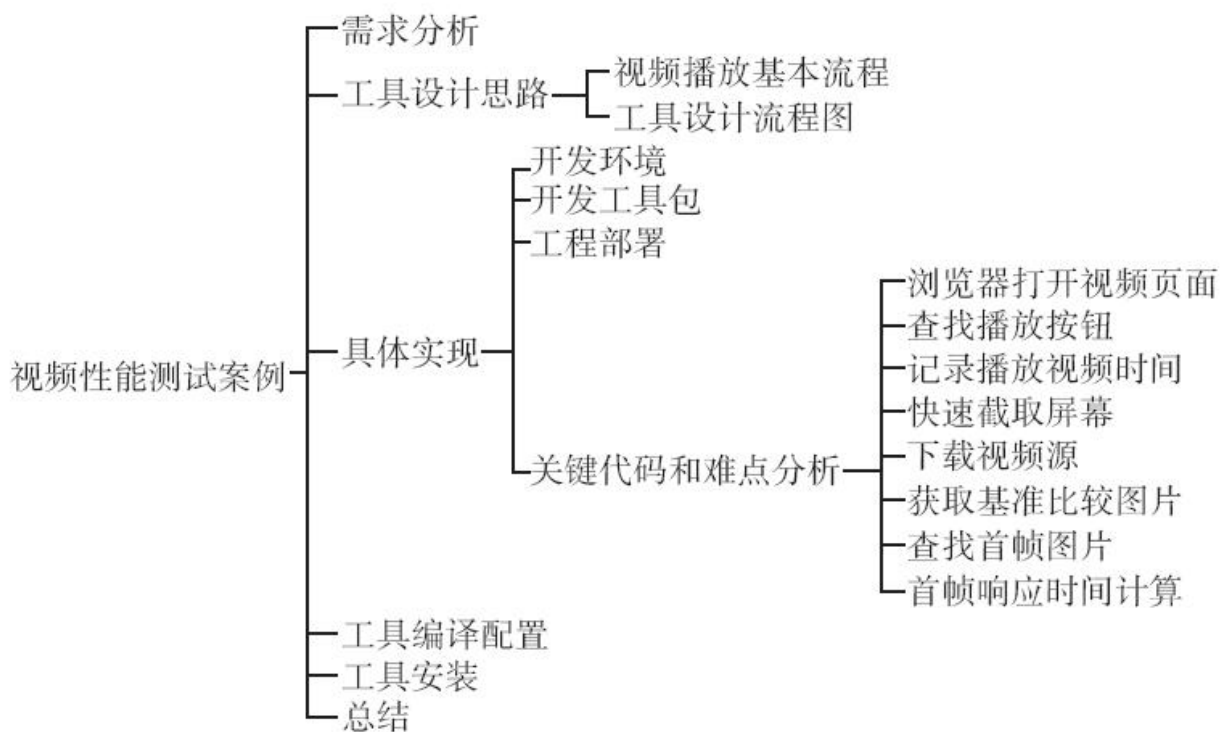


图8-1 本章知识结构图

8.1 视频性能测试需求分析

随着移动终端的快速发展，通过手机观看视频已经成为大众所喜爱的一种休闲方式。当前市场上视频播放器App品种众多（包括优酷、乐视、搜索、爱奇艺客户端、MX Player、暴风影音以及各种浏览器等），功能越来越丰富，界面也越来越人性化。但我们依然无法回避手机本身硬件和软件存在差异的客观现实，这就造成了不同的手机在运行视频播放器程序时有快有慢。在实际播放器项目测试中梳理，影响视频播放体验的因素最终通过以下几个性能指标来体现，如图8-2所示。

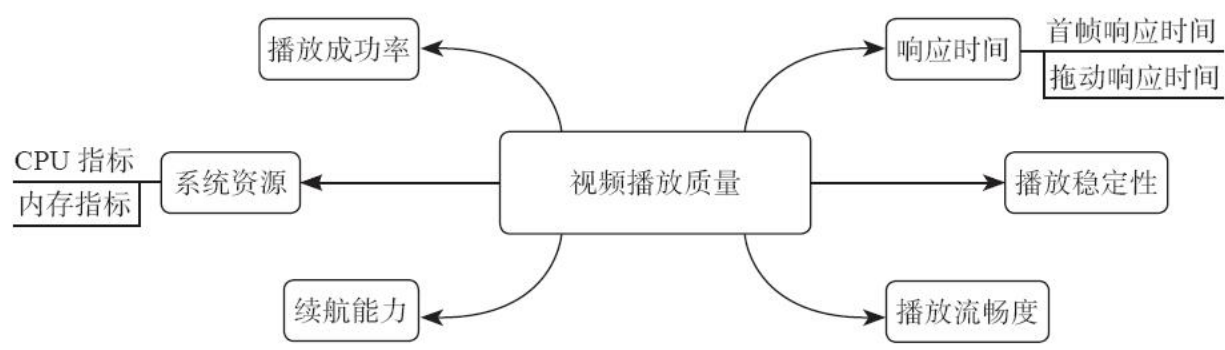


图8-2 衡量视频播放器的相关指标

对于上图中所示各项，分析如下：

·**首帧响应时间**：即用户首次加载视频，获取首个完整关键帧所需的时长，通俗的理解就是从用户点击播放按钮到出现第一帧视频画面所需要的时间。关于这个指标详细的解释请读者参考8.2节。

·**拖动响应时间**：即用户拖动进度条到指定位置后，出现指定位置的第一帧视频所需要的时间。例如当前播放器在1分30秒，拖动进度条到10分钟位置后到出现第10分钟的视频首帧画面所需要的时间就是Seek到10分钟位置的响应时间。

·**播放流畅度**：即视频播放过程1秒钟时间里显示的图片的帧数，也可以理解为图形处理器每秒钟能够刷新几次。帧率越大，画面越流畅；帧率越小，画面越有跳动感。现在市场主流的视频帧率是15帧/秒，超清和蓝光视频是25帧/秒。

·**播放成功率**：即成功播放视频数占总共播放视频总数的比例。这个指标在网络视频播放器衡量中（包括点播和直播）尤其重要，也是衡量播放器好坏的一个非常重要的指标。考虑到测试的片源数量有限，为了更加准确地反映这个指标的有效性，所以通过后台上报来统计播放成功率。例如播放失败片源数为1万，总播放片源数为100万，那么播放成功率即为 $(100-1) / 100 \times 100\% = 99\%$ 。

·**续航能力**：即在手机满电的情况下，持续播放视频待机时长。待机时间越长越好。

·**CPU指标**：即视频播放器在启动和播放视频过程中，CPU所占用的情况。如果CPU占用过高，就会出现手机发烫、续航能力降低的现象。

·**内存指标**：即视频播放器在启动和播放视频过程中，内存所占用的情况。一般内存占用越低越好。

·**播放稳定性**：视频在播放过程中，不会因为播放时间长而导致视频播放质量的下降，主要包括音视频同步、画面的质量、流畅度、响应时间等指标。

上述指标，旨在通过对不同的手机终端进行性能上的考察，增加性能区分维度，为视频播放器测试提供技术保障，从而满足用户的需求。从这些指标上看，视频首帧响应时间尤其重要。因为对于用户来说，播放视频最先体验到的性能指标就是视频播放的首帧响应时间，它的快慢直接关系到用户对于产品的第一印象。所以本章在涉及的视频性能指标中，以QQ浏览器中视频播放器为例，和读者一起分享浏览器中视频播放首帧响应时间的测试方案。

8.2 视频首帧性能测试方案的设计思路

8.2.1 视频播放流程

相信大部分读者都用手机端的浏览器观看过视频，笔者在这里和读者一起了解一下用户浏览器播放视频的基本流程。

(1) 打开浏览器，在视频地址栏中输入视频的URL地址，如图8-3所示。



图8-3 浏览器打开视频网页页面

(2) 点击视频页面中的播放按钮，浏览器解析页面，获取到当前视频的真实片源地址后，播放器请求下载当前视频源，所以从用户的角度可以看到如下的屏幕，如图8-4所示。



图8-4 正在加载视频

(3) 下载到一定大小的视频源后，播放器解码器进行解码，再通过手机屏幕显示出首帧画面，如图8-5所示。



图8-5 出现视频首帧画面

在上述浏览器视频播放基本流程中，从点击播放按钮到出现视频的首帧画面所需要的时间就是前面所说的性能指标的首帧响应时间。

8.2.2 设计思路

前面通过播放流程的方式介绍了视频首帧响应时间的概念，那么是如何计算响应时间的呢？从播放流程可以清楚地知道：只要确定开始时间（也就是点击播放按钮的时间）和结束时间（也就是出现视频首帧的时间）就可以计算出视频的首帧响应时间。如何确定开始时间和结束时间就是本设计的关键，下面简单介绍一下确定的方法。

（1）开始时间：相对而言比较简单，只要记录视频点击播放按钮的系统时间就可以确定。

（2）结束时间：通过截屏记录点击播放按钮到出现视频首帧之间手机整个屏幕画面，然后从记录屏幕画面中找出首次出现视频首帧的图片。由于不同的视频首帧画面是不同的，所以在设计中，需要从播放视频源中提取原始的视频首帧图片作为基准图片，再通过基准图片和截屏图片做比较，找出首帧图片，再把截取到首帧图片的系统时间作为结束时间。

基于上述方案介绍，视频首帧设计流程图如图8-6所示。

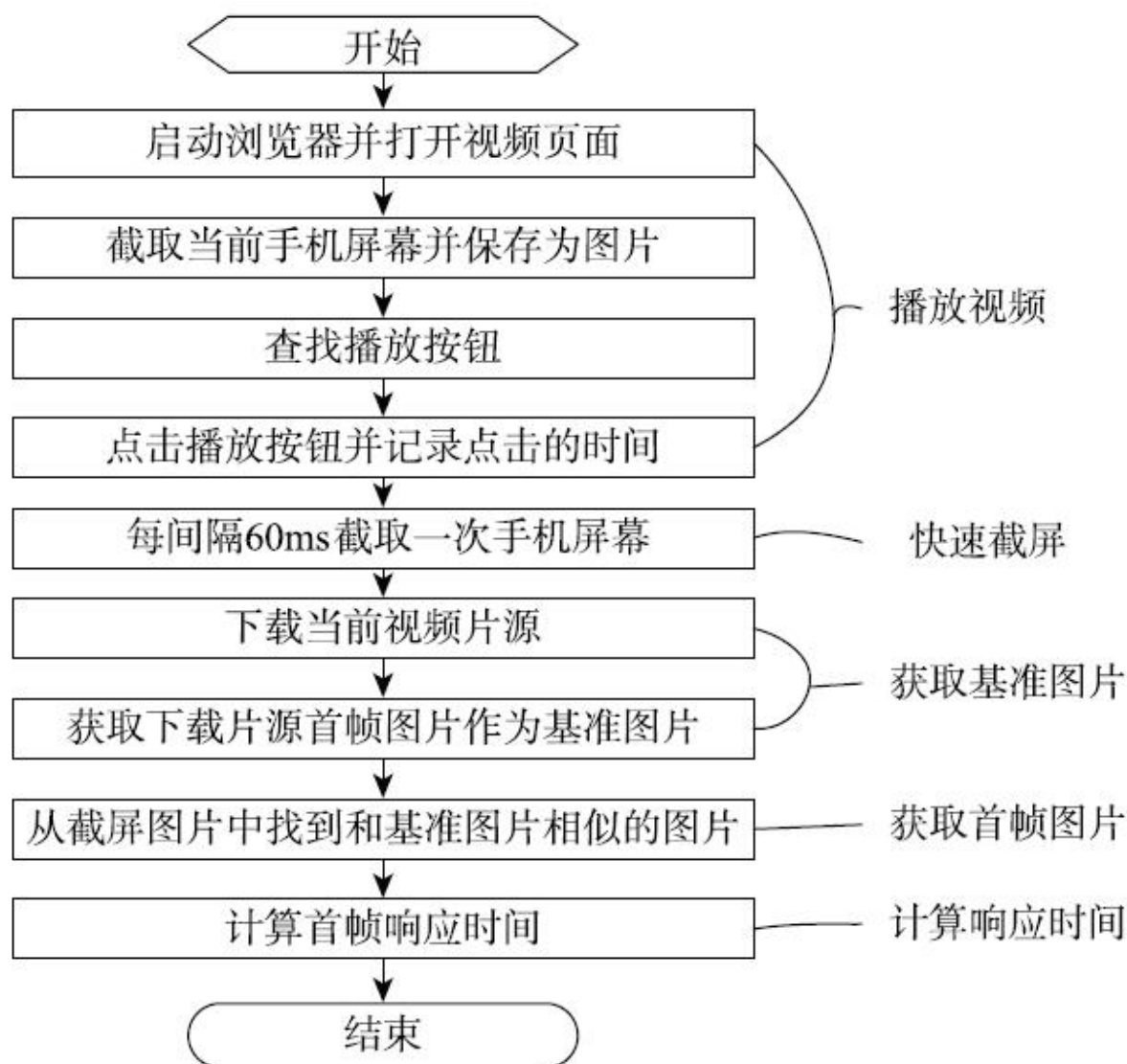


图8-6 视频首帧设计流程图

根据上述流程图，详细介绍一下整个设计实现的基本思路：

·**播放视频**：测试工具通过浏览器打开被测试视频页面，截取当前的手机屏幕作为图片文件保存到SD存储卡中，再通过相关技术查找图片文件上视频播放的位置，然后模拟一个点击播放按钮事件，同时记录当前时间作为点击播放按钮的时间。

·**快速截屏**： 点击播放按钮同时启动快速截取屏幕，每隔60ms秒截取一次屏幕。截取一定张数图片并按照当前截取的系统时间作为文件名一次性写入SD存储卡（例如：截取总张数为120张，因为每张间隔是60ms，这120张图片代表着12s内当前手机屏幕的变化情况）。

·**获取基准图片**： 为了从这120张图片中找出首帧图片，需要下载当前视频源，再利用相关技术从下载视频源中提取视频的原始首帧图片作为基准图片。

·**获取首帧图片**： 把截屏图片按照文件名按从小到大的顺序排列，再把基准图片和它们逐一比较，找到第一个相似度大于90%的截屏图片即为测试要找的首帧图片。

·**计算响应时间**： 因为截屏的图片文件名是以截屏的当前系统时间保存的，所以找到的首帧图片文件名和点击播放按钮时间差就是测试的首帧响应时间。

大家都知道，实际测试中视频首帧响应可能会受到外界不同因素的干扰（如网络、视频服务器等）和手机本身性能的影响。为了消除外界因素的影响，在测试中，需要对多个片源进行测试，然后把它们的平均时间作为响应时间，从而保证测试数据的真实有效性；对于手机本身性能的影响，后面章节详细讨论了选择手机的标准，具体请参考8.3.4节的第四部分。

8.3 视频首帧性能测试方案的具体实现

本节主要基于前面介绍的视频首帧性能测试设计思路，来逐步介绍性能测试工具的开发过程。由于代码比较多，在举例中，只针对主要功能点加以阐述。

8.3.1 开发工具准备

由于性能测试工具是在Android系统开发基础上进行的，所以先了解一下如何搭建工具开发环境，下面以Windows系统为例。

在Windows系统中搭建应用程序开发环境所需要的工具如下：

- Java SDK（建议Java1.6及以上版本）
- Eclipse IDE 4.4及以上版本
- Android SDK（Windows版本R24）
- Android NDK（Windows版本R11）
- ADT插件

关于这些工具包的下载和安装，请参考相关的资料。

8.3.2 测试环境准备

在前面介绍的设计思路中，涉及截屏、获取下载片源首帧图片、查找播放按钮以及图片相似度比较等功能需要系统源码和相关开源库的支持，对于系统源码这里以Android 4.4.4系统为例进行阐述。在整个开发过程中，需要的资源包如下：

- OPENCV SDK（2.4.10版本）

- FFMPEG库

- Android相关的源代码头文件

- Android系统相关的动态库

依据下列介绍准备与步骤相关的工具包。

1.OPENCV SDK

OPENCV（Open Source Computer Vision Library）是一个基于（开源）发行的跨平台计算机视觉库，可以运行在Linux、Windows、Mac OS以及Android操作系统上。在本章实例中，需要用OPENCV SDK查找播放按钮和比较相似图片。相关的SDK可从OPENCV官网下载Android版，然后依照官方相应操作步骤把OPENCV SDK导入Eclipse中。

2.FFMPEG库

FFMPEG是一套可以用来记录、转换数字音频、视频，并能将其转化为流的开源计算机程序。本章实例需要通过获取下载视频源的原始首帧图片作为比较的基准图片，读者可以从FFMPEG官网下载代码，然后在Ubuntu系统上配置NDK环境，再通过编译就可以得到FFMPEG库和相关的头文件。

3.Android相关的源代码头文件

在测试中所涉及的截屏功能，是通过Android系统相关接口实现的。为了保证编译通过，我们需要从官网上下载Android系统部分源代码头文件，在这里下载的Android4.4.4系统头文件目录如图8-7所示。



development	2015/8/7 16:44	文件夹
external	2015/8/7 16:44	文件夹
frameworks	2015/8/7 16:44	文件夹
hardware	2015/8/7 16:44	文件夹
system	2015/8/7 16:44	文件夹

图8-7 系统相关头文件目录

4.Android系统相关的动态库

工具编译成功后，为确保能够正确链接通过，这里还需要准备Android系统相关的动态库，需要的动态库文件在Android系

统/system/lib目录下。因为前面下载的是Android 4.4.4系统的头文件，这里需要对应的动态库也必须是4.4.4系统版本的，图8-8所示为需要复制的动态库文件。

 libbinder.so	2015/4/10 10:44	SO 文件	174 KB
 libgui.so	2015/4/10 10:44	SO 文件	294 KB
 libskia.so	2015/4/10 10:44	SO 文件	2,102 KB
 libui.so	2015/4/10 10:44	SO 文件	42 KB
 libutils.so	2015/4/10 10:43	SO 文件	82 KB

图8-8 需要复制的动态库文件

8.3.3 工程部署

前面已搭建好了开发环境并准备了相关工具包，下面就可以开始按设计思路开发测试工具。

1.创建Android工程

在eclipse中，创建一个Android的应用程序，取名为prefVideo工程，并给该工程添加一个native环境，如图8-9所示。

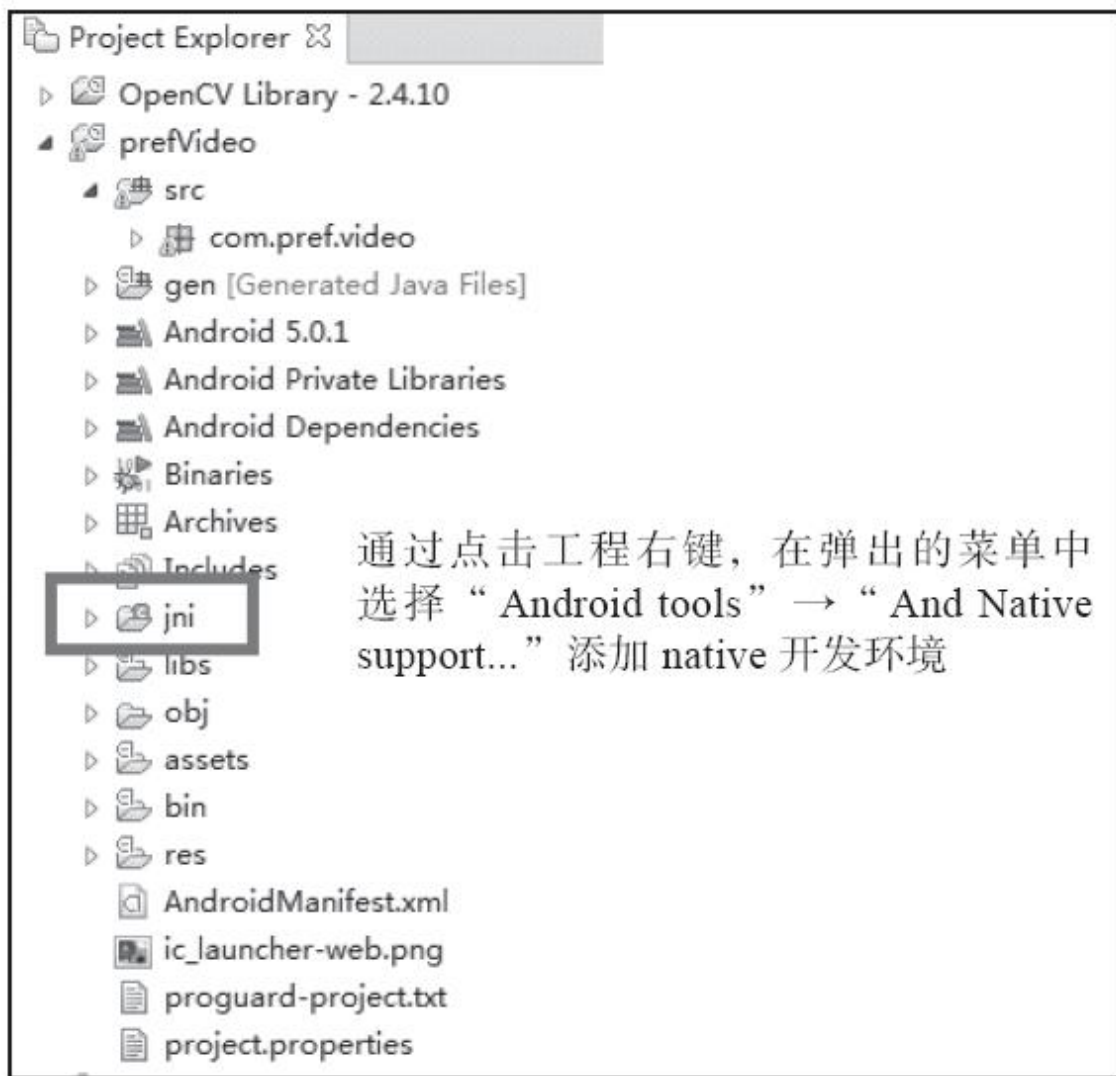


图8-9 创建Android工程

2.配置开发工具包

前面把准备好的相关工具包导入测试工具工程，具体的步骤如下：

(1) **导入OPENCV SDK** 。通过点击工程右键，在弹出的菜单中选择“properties”选项，弹出下列对话框，再按照对话框上标号数字的

顺序把OPENCV SDK导入perfvideo工程，如图8-10所示。

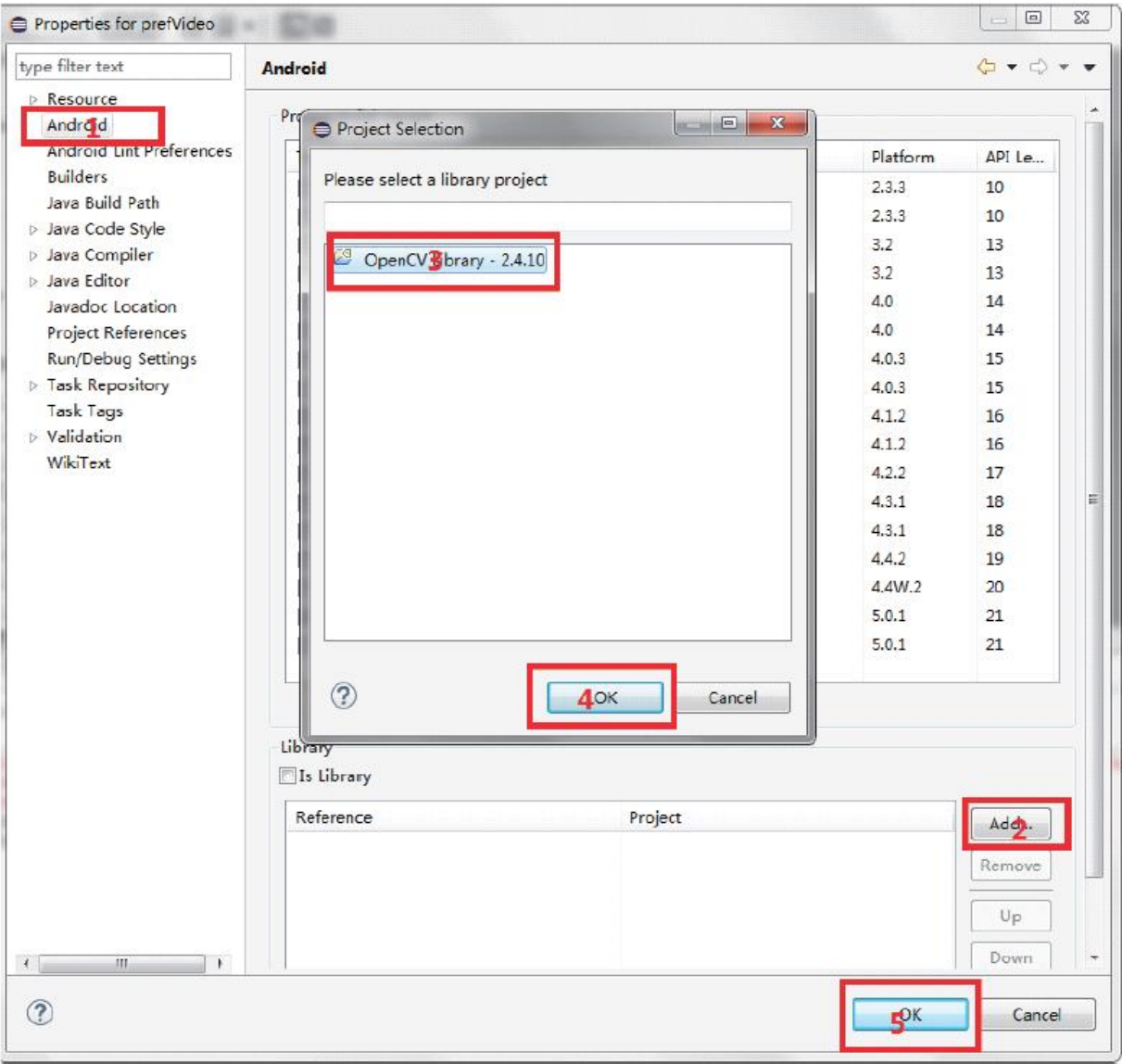


图8-10 导入OPENCV SDK

(2) 导入其他工具包。在工程jni目录下创建Android和FFMPEG目录，然后把前面下载的Android系统的头文件和库文件复制到Android

目录；把FFMPEG头文件和库文件复制到FFMPEG目录，如图8-11所示。

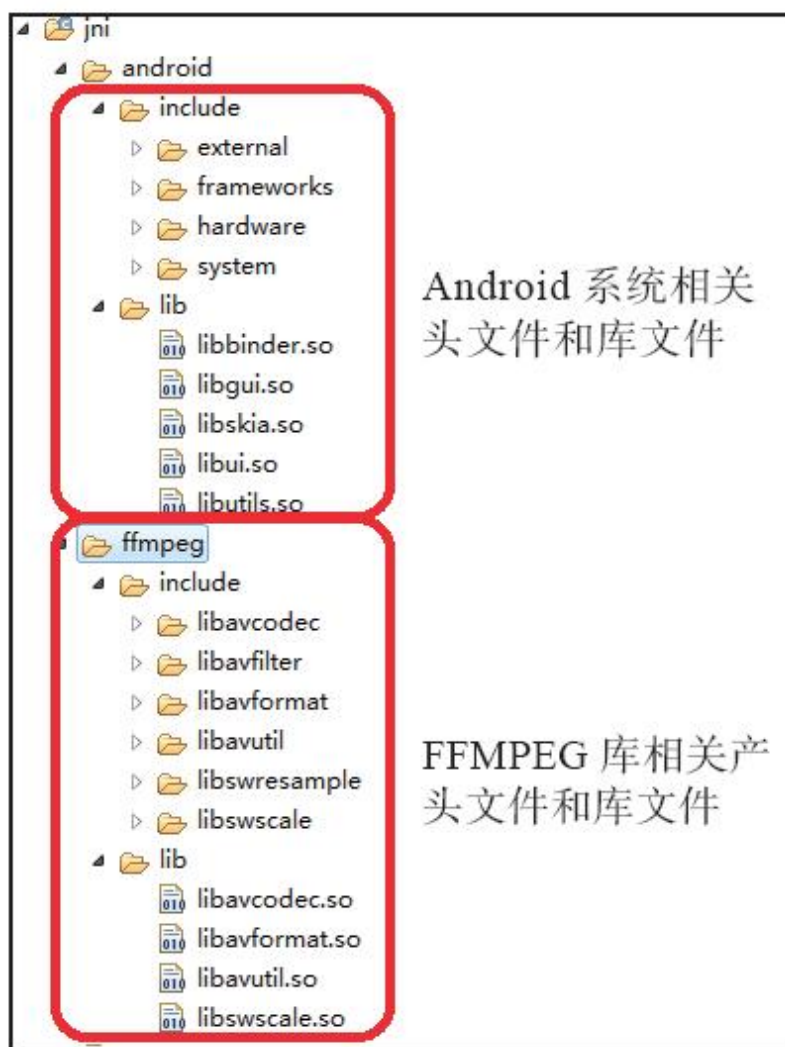


图8-11 导入其他相关的头文件和库文件

8.3.4 关键代码和难点分析

下面我们就根据上面的设计思路，依次来分析下列代码。

1.启动浏览器打开视频网页

在Android开发中，Intent机制是一种运行时绑定（run-time binding）机制，它能在程序运行过程中连接两个不同的组件。通过Intent，程序可以向Android表达某种请求或者意愿，Android会根据意愿的内容选择适当的组件来完成请求。例如，本例中测试工具希望打开浏览器访问视频页面，那么这个应用只需要发出ACTION_VIEW给Android，Android就会根据Intent的请求内容，启动浏览器并打开视频页面，具体实现如代码清单8-1所示。

代码清单8-1 启动浏览器并打开视频页面

```
Intent intent = new Intent();    //创建一个
Intent
intent.setAction(Intent.ACTION_VIEW);    // 设置执行动作

intent.setData(Uri.parse("视频地址
"));    //设置打开视频页面的
URL
intent.setClassName("com.tencent.mtt","com.tencent.mtt.SplashActivity");
//com.tencent.mtt 浏览器包名

com.tencent.mtt.SplashActivity 浏览器主
Activity名字
```



```
startActivity(intent);
```

在上面的代码中，关于浏览器包名和主Activity可以通过Android SDK的build-tools目录下的aapt命令来获取（最新SDK下aapt命令和被测APP可能存在兼容性问题，建议用21.x目录下的aapt命令），具体如图8-12所示。

```
C:\>aapt dump xmltree qqbrowser.apk AndroidManifest.xml
N: android=http://schemas.android.com/apk/res/android
E: manifest <line=164>
  A: android:versionCode(0x0101021b)=<type 0x10>0x9a470
  A: android:versionName(0x0101021c)="6.3.0.1920" <Raw: "6.3.0.1920">
  A: android:installLocation(0x01010217)=<type 0x10>0x0
  A: package="com.tencent.mtt" <Raw: "com.tencent.mtt">          包名
E: permission <line=170>
  A: android:name(0x01010003)="com.tencent.mtt.broadcast" <Raw: "com.tencent.mtt.broadcast">
  A: android:protectionLevel(0x01010009)=<type 0x11>0x2
E: uses-permission <line=175>
  A: android:name(0x01010003)="com.tencent.mtt.broadcast" <Raw: "com.tencent.mtt.broadcast">
E: uses-permission <line=176>
  A: android:name(0x01010003)="android.permission.WAKE_LOCK" <Raw: "android.permission.WAKE_LOCK">
E: uses-permission <line=177>
  A: android:name(0x01010003)="android.permission.ACCESS_COARSE_LOCATION" <Raw: "android.permission.ACCESS_COARSE_LO
CATION">
E: uses-permission <line=178>
  A: android:name(0x01010003)="android.permission.ACCESS_WIFI_STATE" <Raw: "android.permission.ACCESS_WIFI_STATE">
E: activity <line=423>
  A: android:theme(0x01010000)=0x7f0f0006
  A: android:label(0x01010001)=0x7f0c000a
  A: android:name(0x01010003)="com.tencent.mtt.SplashActivity" <Raw: "com.tencent.mtt.SplashActivity">          主Activity名
  A: android:alwaysRetainTaskState(0x01010203)=<type 0x12>0xffffffff
  A: android:windowSoftInputMode(0x0101022b)=<type 0x11>0x22
E: intent-filter <line=429>
  E: action <line=430>
    A: android:name(0x01010003)="android.intent.action.MAIN" <Raw: "android.intent.action.MAIN">          通过MAIN找出APP主Activity
  E: category <line=432>
    A: android:name(0x01010003)="android.intent.category.LAUNCHER" <Raw: "android.intent.category.LAUNCHER">
  E: category <line=433>
    A: android:name(0x01010003)="android.intent.category.MULTIWINDOW_LAUNCHER" <Raw: "android.intent.category.MU
LTIWINDOW_LAUNCHER">
```

图8-12 获取包名和主Activity

2.查找播放按钮

对于不同的视频网站，如何查找播放按钮所在的位置呢？首先看一下浏览器打开的各个视频网站的页面。图8-13所示为各大视频网站播放页面在手机上的截图。

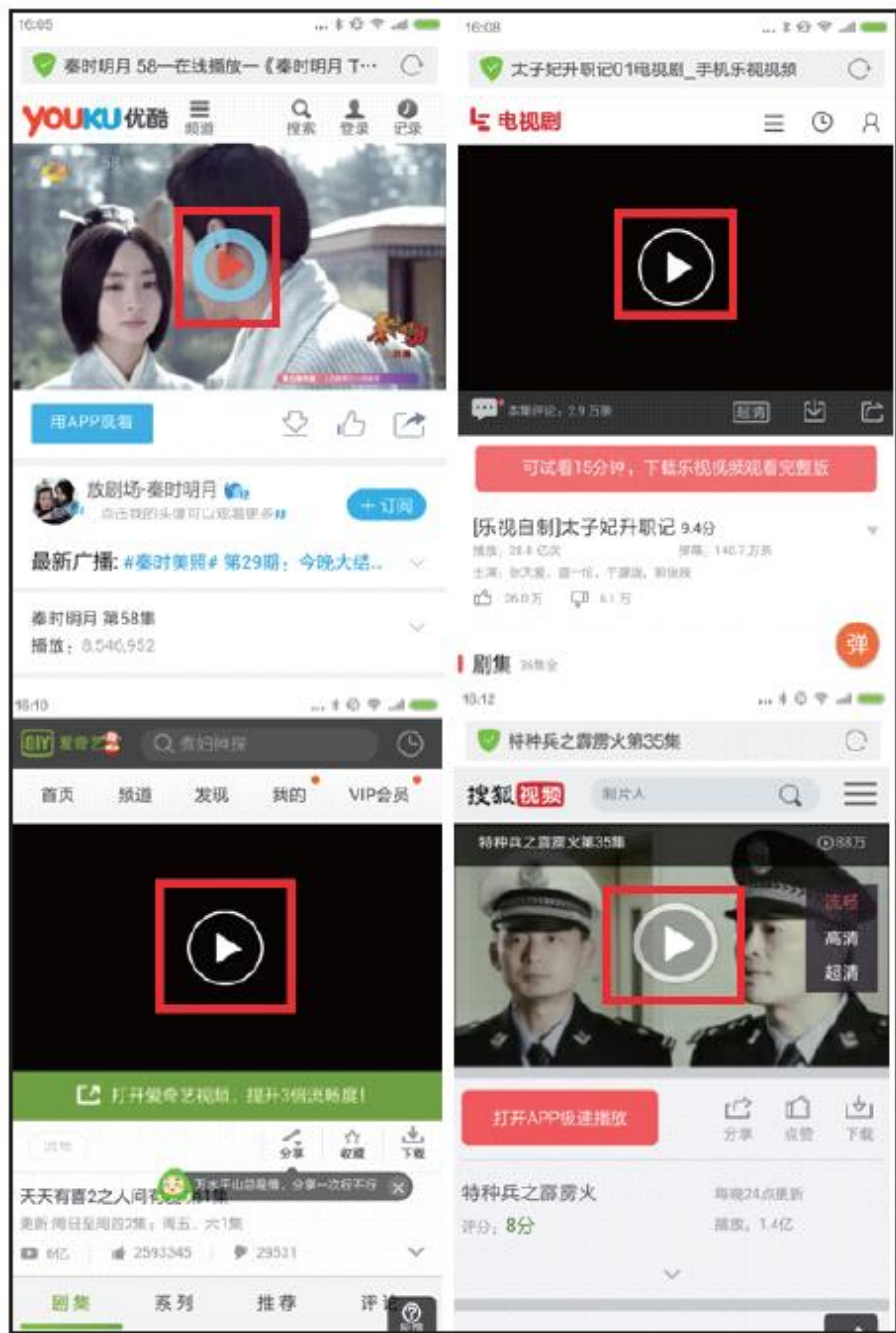


图8-13 各大视频网站播放页面

从图8-13中我们可以看到，各大视频网站页面中的视频播放按钮有一个明显的共同特征：播放按钮都是圆形的。由于图片是按照手机相同比例截取的，那么只要找到圆形在图片中的位置就可以获取播放按

钮的位置。获取圆形在图片中的位置方案很多，这里利用开源
OPENCV SDK来查找圆形在图片中的位置，其具体实现如代码清单8-2
所示。

代码清单8-2 查找播放按钮

```
Mat dst = new Mat();
//从

SD卡中读取当前截屏的屏幕图片

Mat source=Highgui.imread("截取的当前手机屏幕图片
",1);
final List<MatOfPoint> rawContours = new ArrayList<MatOfPoint>();
final Mat hieararchy = new Mat();
//把原图像转化成灰度图像

Imgproc.cvtColor(source, source, Imgproc.COLOR_BGR2GRAY );
//使用高斯平滑进行模糊降噪

Imgproc.GaussianBlur(source, dst, new Size(11, 11), 0, 0);
//运行

Canny算子

Imgproc.Canny(source, dst, 80, 240, 3, true);
//查找图片中的各种轮廓

Imgproc.findContours(dst,rawContours,hieararchy,Imgproc.RETR_EXTERNAL,
                    Imgproc.CHAIN_APPROX_SIMPLE);
for (int i = 0; i < rawContours.size(); i++) {
    MatOfPoint2f point2f = new MatOfPoint2f();
    //改变当前轮廓

    Mat的属性

    rawContours.get(i).convertTo(point2f, CvType.CV_32FC2);
    MatOfPoint2f approxCurve = new MatOfPoint2f();
    //计算当前轮廓的凸包，凸包也就是图形上的各个点

    Imgproc.approxPolyDP(point2f, approxCurve,Imgproc.arcLength(point2f, true) * 0.02,
    true);
    //查找当前轮廓面积是否大于
```

20000个像素的圆

```
if ((Math.abs(Imgproc.contourArea(approxCurve)) > 20000)
    && (!Imgproc.isContourConvex(rawContours.get(i)))) )
{
    List<Point> isConvex = new ArrayList<Point>();
    Converters.Mat_to_vector_Point(approxCurve, isConvex);
    //如果当前轮廓凸包个数是
```

3, 就是三角形; 个数是

4, 就是四边形, 依次类推。这里我们认为个数大于

6时, 是一个圆形。

```
    if(isConvex.size() > 6)
    {
        //计算圆形的圆点的坐标位置

        Rectr = Imgproc.boundingRect(rawContours.get(i));
        radius[0] = (r.x + r.width / 2);
        radius[1] = (r.y + r.height / 2);
        return radius;
    }
}
```

OPENCV SDK提供了一个Houghcircles函数, 也可以利用这个函数直接获取圆形, 具体的读者可以参考与OPENCV SDK相关的文档。另外代码中涉及当前屏幕截取可以调用Android系统自带的screencap命令实现。

3.播放视频并记录当前开始时间

关于模拟一个事件点击, 相信网上介绍的方法很多。但是这些方法有一个共同的特点: 很难准确地记录时间的点击时间。为了解决这个问题, 笔者在这里采用了input事件注入原理。首先我们来了解一下

Android输入事件流程，在Android系统中，input事件处理流程如图8-14所示。

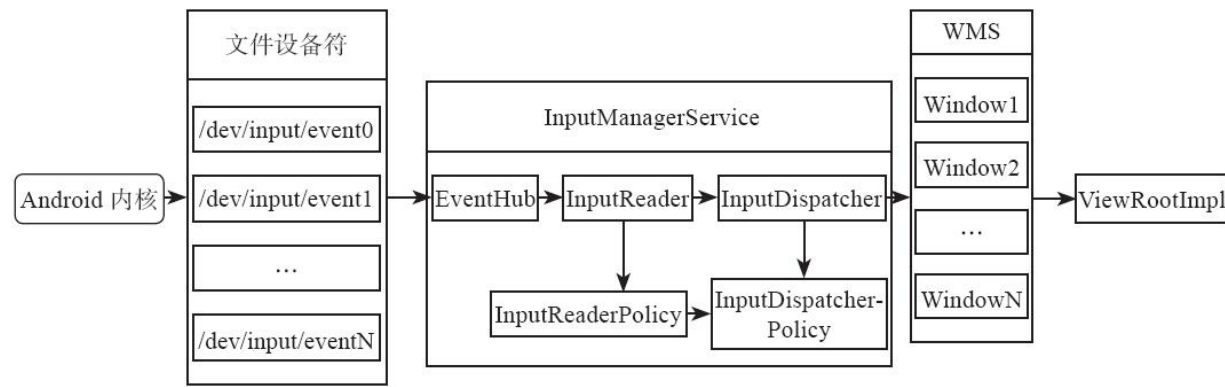


图8-14 input事件处理流程

Android内核接受输入设备的中断，并将原始事件的数据写入设备节点中，然后由InputManagerService从文件设备中取出相应的事件；WindowManagerService（以下简称WMS）服务运行在SystemServer进程中，应用程序启动Activity时，需要请求WMS为启动的Activity创建对应的窗口，而WMS启动之后，经逐层调用，把读取输入事件传递到Java层；ViewRootImpl和WMS之间的通信是通过Binder机制实现的，ViewRootImpl获取到响应的事件再分发DecorView（Activity根视图），并由它分发到相应的View，这是Android输入事件简单处理流程。如果读者感兴趣的话，可以参考相关的书籍和Android系统的源代码。

从上面的Android输入事件处理机制，可以很清楚地知道手机硬件通过内核驱动程序把用户操作相应事件写入文件设备符中，然后由Android系统从文件设备符中读取事件，再通过层层分发的机制分发给

我们的应用程序。为了更好地了解Android系统事件机制，这里再看一下Android系统提供的getevent与sendevent两个工具供使用者从文件设备符中直接读取输入事件或写入输入事件。这里以小米3为例，通过Android系统的getevent命令来获取一下touch屏幕某个位置所发生的系统事件序列，如图8-15所示。

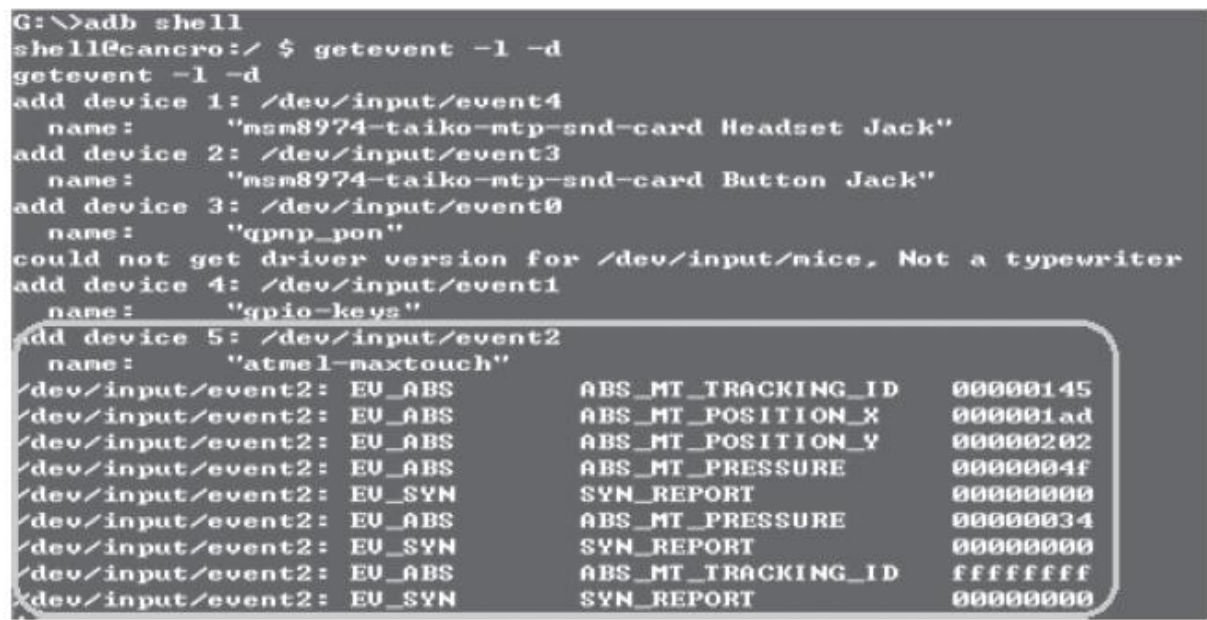


图8-15 getevent获取touch事件

从图8-15中可以看到屏幕的touch是通过文件设备符/dev/input/event2实现的，只要向event2文件设备符注入播放按钮位置touch事件，就可以实现测试的要求。这里对于Android系统的sendevent做了一些改造，其具体实现如代码清单8-3所示。

代码清单8-3 模拟点击事件


```

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/input.h>
unsigned long long getSystemTime()
{
    //计算系统当前的时间

```

, 单位为毫秒。其功能和

Java的

System.currentTimeMillis()相同

```

struct timeval tv;
gettimeofday(&tv, NULL);
return((unsigned long long)tv.tv_sec)*1000 + ((unsigned long
long)tv.tv_usec)/1000;
}
//这里实现了一个点击屏幕任何位置的接口, 其中

```

x,y 就是我们需要点击屏幕的坐标位置

```

void touchXY(int x ,int y)
{
    struct input_event event;
    //打开文件设备符

    int fd = open("/dev/input/event2", O_RDWR);
    //打开失败之间返回

-1
    if(fd < 0) {
        return ;
    }
    //定义一个

```

touch事件的序列

```

    int typevalue[] = {EV_ABS, EV_ABS, EV_ABS, EV_ABS, EV_ABS, EV_SYN, EV_ABS, EV_SYN};
    int codevalue[] = {ABS_MT_TRACKING_ID, ABS_MT_POSITION_X, ABS_MT_POSITION_Y,
ABS_MT_PRESSURE, ABS_MT_TOUCH_MAJOR, SYN_REPORT, ABS_MT_TRACKING_ID, SYN_REPORT};
    int eventvalue[] = {0x000000145, 0x0000001ad, 0x000000202, 0x00000004f,
0x000000034, 0x000000000, 0xffffffff, 0x000000000};
    for(int i = 0; i < sizeof(typevalue)/sizeof(int); i++)
    {
        //给事件各个变量清

0
        memset(&event, 0, sizeof(event));
        event.type = typevalue[i];
        event.code = codevalue[i];
        switch(event.code)
        {
            case ABS_MT_POSITION_X:
                //把当前

```

touch的位置坐标

x赋值给当前注入的事件

```
        event.value = x;
break;
case ABS_MT_POSITION_Y:
    //把当前
```

touch的位置坐标赋值给当前注入的事件

```
        event.value = y;
break;
default:
    event.value = eventvalue[i];
break;
}
//把事件序列写入文件设备符
```

```
if(write(fd, &event, sizeof(event)) < (ssize_t) sizeof(event)) {
break;
}
}
close(fd);
//打印当前
```

touch的位置的时间

```
    printf("%lld",getSystemTime());
}
```

上面的代码仅定义了一个touch屏幕的接口，而调用是在批量截屏时实现的。另外这里还有一点要说明的是，不同机型通过getevent获取的touch事件的事件序列可能不一样。相信有读者就会产生疑问：“是不是针对于不同的机型都需要修改源代码？”而笔者实际在工作中利用getevent获取的事件序列写入文件中，然后代码再从文件中读取事件序列值后赋值给typevalue、codevalue、eventvalue变量（三个变量具体含义请参考系统源码inPut_event结构），这样设计就具有一定的通用性，关于代码读者可以自行实践。

4.快速截取屏幕

笔者前期开发性能测试工具的过程，采用录像分析帧进行了尝试。但是实际使用中却遇到了很多问题，例如摄像头清晰度、摄像之间的距离等外部原因，最主要的是由于视频播放是动态的，摄像视频在分帧以后，图片非常模糊。在这种情况下，只能靠人眼分析，工具很难准确地比较识别。鉴于此，笔者阅读了Android系统自带的screencap命令的源码，发现它是调用SurfaceFlinger提供的截屏接口ScreenshotClient，而ScreenshotClient通过进程间通信调用SurfaceFlinger service的截屏功能。这里为了满足性能测试的需求，笔者基于screencap命令源代码对于截屏做了以下几个方面的优化。

- 截取的当前屏幕通过struct结构直接存储在内存上，而不是立即写到文件上。等截屏达到一定数量的图片后，以每次截屏的时间作为文件名一次性写入SD存储卡。

- 对截取的当前屏幕做了一定比例的缩小，笔者实际缩小的比例是0.3倍。

- 批量截屏的每次截屏间隔时间是60ms。

通过上述的优化，首先避免在截屏中过多写入SD存储卡而影响测试的性能；其次把当前屏幕做一定缩小，不仅可以缩短截屏时间，而且还可以节省存储到内存的空间；最后每次截屏间隔60ms，主要考虑

当前市场上多数的视频帧率在15帧/秒左右，也就是每帧的时间间隔在67ms左右。具体实现如代码清单8-4所示。

代码清单8-4 快速截屏

```
//引用

Android命名空间，以便后面代码能直接调用

Android命名空间的函数

using namespace android;
/*定义一个图片内存临时保存的

Struct结构

cur_time 保存截取当前图片的系统时间

pPic 图片保存的内存地址

height 图片的高

width 图片的宽

*/
struct Pic_Map
{
    unsigned long long cur_time;
    void *pPic;
    struct Pic_Map * next;
    int height;
    int width;
    int format ;
};
struct Pic_Map *pmap = NULL;
static void saveFrameToBuffer(const void* base,int width,
    int height,int strige,int bytesPerPixel,int format)
{
    if(base !=NULL)
    {
        struct Pic_Map *ptmp = new struct Pic_Map;
        //分配一个内存空间来保存当前截屏的图片

        ptmp->pPic = new uint8_t[height*width *bytesPerPixel];
        if (ptmp->pPic == NULL)
        {
            delete ptmp;
```

```

        return ;
    }
    //把当前

```

ScreenshotClient获取的当前屏幕图片复制到分配的内存空间中

```

        for (int i = 0; i < heigth ; i ++){
            memcpy((void*)ptmp->pPic + i * width* bytesPerPixel,
                (void*)(base + i * strige * bytesPerPixel), width*bytesPerPixel);
        }
        //保存图片的相关参数

        ptmp->width = width;
        ptmp->heigth = heigth;
        ptmp->format = format;
        //保存截屏的当前时间

        ptmp->cur_time = getSystemTime();
        ptmp->next = pmap;
        pmap = ptmp;
    }
}
/*通过

```

SkBitmap类把图片保存到文件中

```

*/
static void saveImage(void *base ,const char * fname,int width ,int heigth)
{
    SkBitmap b;
    b.setConfig(SkBitmap::kARGB_8888_Config, width, heigth);
    b.setPixels(base);
    SkImageEncoder::EncodeFile(fname, b,
        SkImageEncoder::kPNG_Type, SkImageEncoder::kDefaultQuality);
}
/*把

```

struct结构中的图片批量保存到文件中

```

*/
static void saveBufferToImage(const char *dir)
{
    char path[256];
    do
    {
        struct Pic_Map *ptmp = pmap;
        if (ptmp)
        {
            pmap = ptmp->next;
            //以当前截屏的时间作为文件名保存图片

            sprintf(path,"%s/%lld.png",dir,ptmp->cur_time);
            saveImage((void*)ptmp->pPic,path,ptmp->width,ptmp->heigth);
            //释放临时存放图片的内存空间

```

```

        if(ptmp->pPic)
            delete ptmp->pPic;
        delete ptmp;
    }
    else
    {
        break;
    }
}while(true);
}
int main(int argc, char** argv)
{
    //把传入的 (
    x,y) 坐标转换为整形

    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    //相关变量定义

    void const* base = 0;
    uint32_t w = 0,h = 0,s,f;
    size_t size = 0;
    ScreenshotClient screenshot;
    //初始化

    ProcessState::self()->startThreadPool();
    sp<IBinder> display = SurfaceComposerClient::getBuiltInDisplay(
        ISurfaceComposer::eDisplayIdMain);
    if (display !=NULL)
    {
        status_t rv = UNKNOWN_ERROR;
        for (int i = 0; i< count + 1; i++)
        {
            unsigned long long cur_start = getSystemTime();
            //获取当前屏幕

            if ( (w == 0 )|| (h == 0))
            {
                rv = screenshot.update(display);
            }
            else
            {
                rv = screenshot.update(display,w,h);
            }
            if (rv == NO_ERROR)
            {
                if (i == 0)
                {
                    //获取当前屏幕的宽和高后，计算缩放后的宽和高的大小

                    w = screenshot.getWidth() * 0.3;
                    h = screenshot.getHeight() * 0.3;
                }
                else
                {

```

```

//把当前的屏幕保存到临时内存

struct中

saveFrameToBuffer(screenshot.getPixels(),screenshot.getWidth(),

screenshot.getHeight(),screenshot.getStride(),bytesPerPixel(screenshot.getFormat()
),screenshot.getFormat());
    }
}
screenshot.release();
//从截取第三张图片开始点击播放按钮，这样可以避免截取第一张图片耗时比较多而影响测试的性能

if (i == 3)
{
    touchXY(x,y);
}
unsigned long long end_start = getSystemTime();
//如果当前截屏时间小于

60ms，挂起当前线程

if(end_start - cur_start < 60* 1000)
{
    usleep(60 * 1000 - (end_start - cur_start));
}
}
//把

struct结构的图片保存到

SD卡上

saveBufferToImage(fname);
return true;
}

```

细心的读者也许会问，这种批量快速截屏是否会影响测试的性能？笔者在设计工具的前期在几款不同的机型上做了大量实验，与录屏分帧的方法做了对比，两种测试的结果基本相同，误差大概在 $\pm 5\%$ 。从这几款实验的机型中笔者发现，如果截屏所耗用的时间小于每次截屏的间隔时间的1/3，是不影响性能测试结果的，在这里读者也可以结合自己的项目验证一下。

这里以小米3为例。在QQ浏览器页面播放视频的情况下，每隔60ms截取一次当前屏幕，截取当前屏幕的耗时平均在12ms，图8-16所示为多次截取屏幕的每次耗时情况。

```
capture picture consume time:9 ms
capture picture consume time:8 ms
capture picture consume time:9 ms
capture picture consume time:17 ms
capture picture consume time:14 ms
capture picture consume time:15 ms
capture picture consume time:9 ms
capture picture consume time:12 ms
capture picture consume time:14 ms
capture picture consume time:16 ms
capture picture consume time:15 ms
capture picture consume time:22 ms
capture picture consume time:5 ms
capture picture consume time:5 ms
capture picture consume time:8 ms
capture picture consume time:10 ms
capture picture consume time:6 ms
capture picture consume time:13 ms
capture picture consume time:7 ms
```

图8-16 多次截取屏幕的每次耗时情况

从图8-16中可以看出，每次截取当前屏幕所消耗时间的资源比较少，而且目前市场上大多数手机配置比小米3手机低的并不多，所以这种方法就能满足市场上主流手机测试的需求。

5. 下载视频

在前面快速截屏部分，工具已经完成了视频播放过程的截屏，批量截屏图片如图8-17所示。

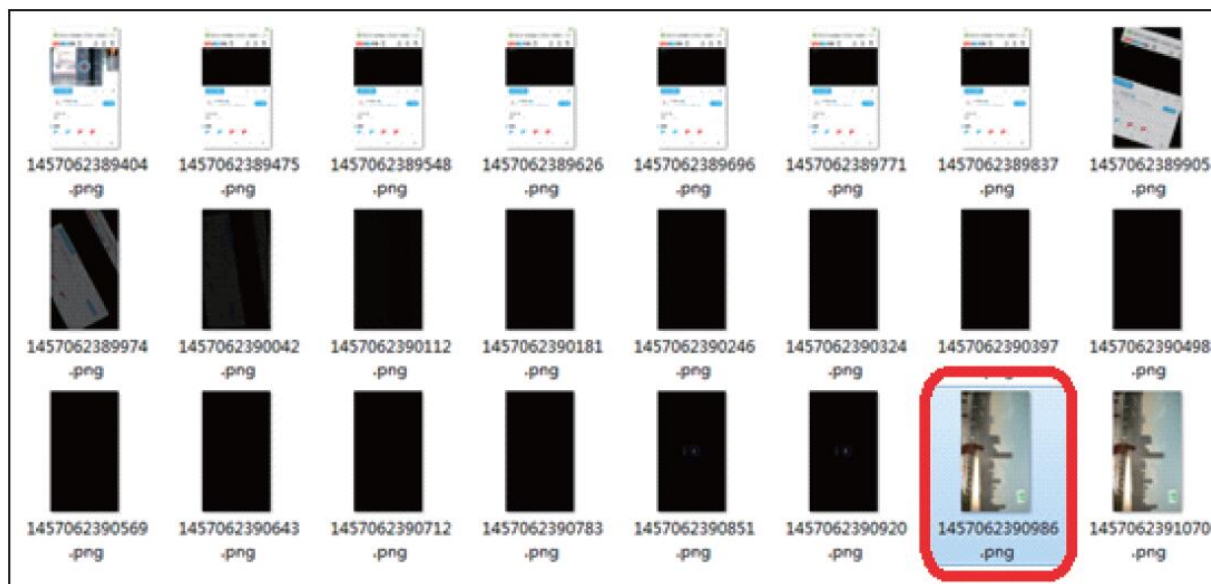


图8-17 批量截屏图片

从图中能很快辨认出选中的图片就是需要找的首帧图片。但是如何才能利用工具自动识别呢？为了解决这个问题，先来简单了解一下视频原理：视频其实也是由连续变化的图片（称为帧）组成的，视频在播放过程中就是把这些连续图片平滑显示出来，利用人的视觉短暂原理，因为人眼无法辨别单幅静态画面。所以该播放片源首帧其实也是一张图片，只要提取片源首帧原始作为基准图片，再和截屏图片逐一比较，就可以找出截屏中的首帧图片。为了方便提取片源首帧原始图片，这里就需要下载当前视频片源。怎么才能获取视频的真实地址呢？这里利用Android系统的Webview打开当前视频片源页面，然后通

过JavaScript代码注入的方式获取当前片源的视频地址源，例如当前打开的一个视频页面后在HTML页面中所看到的真实地址如图8-18所示。

其中Video标签属性src值就是播放视频源的真实地址。而HTML所有页面元素的页面属性都可以通过JavaScript获取，所以这里可以通过Java和JavaScript之间的通信获取Video标签的src属性，具体实现如代码清单8-5所示。



图8-18 HTML页面视频真实地址

代码清单8-5 获取视频源地址

```
//创建一个
webview对象

WebView webV = new WebView(getApplicationContext());
//设置当前
webview支持

javascript
webV.getSettings().setJavaScriptEnabled(true);
//向

webview注入一个

java对象，通过这个对象的

javascript可以返回相关的结果

webV.addJavascriptInterface(new Object(){
```



```

        public String getResult(String vAddr)
        {
            return vAddr;
        }
    }, "JS_RESULT");
    //注入

    javascript脚本，然后通过注入

    java对象的

    getResult返回

    javascript结果

    webV.loadUrl("javascript:JS_RESULT.getResult(document.getElementsByTagName
    ('video')[0].src)");

```

上述实现方式是JavaScript代码获取视频真实地址的一种简单方式，但是随着各大网站的升级和维护，可能要针对不同网站开发不同的JavaScript脚本。笔者在实际工作中使用tcpdump源代码方案替代这种方案，这种方案的基本原理就是打开视频网页之后，监听网络数据包，然后在代码中分析数据包，找出播放视频的真实地址。下面是播放某一网站视频后，通过tcpdump抓取的数据包，如图8-19所示。

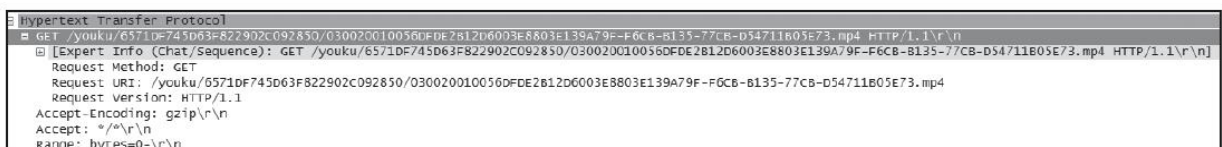


图8-19 tcpdump抓取的视频真实地址

图8-19中选中的部分就是播放器播放视频的真实地址，所以在代码中只要分析监听网络数据包，就能获取视频地址。但是采用这种方式实现，相对来说代码量大，过滤视频地址处理比较复杂。所以有兴趣的读者可以结合自己的项目尝试一下。

接下来利用视频的真实地址下载该片源，视频下载这里是通过Android系统的DownloadManager来实现的，具体实现如代码清单8-6所示。

代码清单8-6 下载视频

```
// 创建一个
DownloadManager
DownloadManager downloadManager = (DownloadManager)
    getSystemService(Context.DOWNLOAD_SERVICE);
// vAddr 是
    javascript返回的视频真实地址

Uri uri = Uri.parse(vAddr);
DownloadManager.Request request = new Request(uri);
// 设置当前下载片源地址和文件名

request.setDestinationInExternalPublicDir("prefvideo", "video.mp4");
// 请求下载当前片源

downloadManager.enqueue(request);
```

上面的代码完成了视频下载的基本功能，笔者发现，现在市场上好多主流视频播放器都支持预缓存功能。也就是在播放当前片源的时候，播放器已经把当前播放片源预先缓存到手机的存储卡上的某个位置，如果该App实现了这样的功能的话，我们就可以省略下载视频源这一步，直接用缓存的视频源。



注意 现在部分视频网站的视频地址源是m3u8格式的，所以下载这类的片源，要先解析m3u8文件，再进行视频源的下载。

6.获取比较基准图片

这里利用FFMPEG库从下载源中提取当前视频的第一帧图片作为基准图片，但这里包括两部分。

- JNI部分：利用FFMPEG库解码当前视频片源，提取视频的首帧。

- Java部分：JNI部分通过Java的bitmap类创建图片文件，然后再通过Java的自定义的函数把视频首帧写入图片文件中。

下面分别列取了JNI部分和Java部分的代码。具体实现如代码清单8-7和代码清单8-8所示。

代码清单8-7 JNI提取视频首帧图片

```
#include <jni.h>
#include <wchar.h>
#include<android/bitmap.h>
//导入

FFMPEG相关头文件

extern "C"{
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libswscale/swscale.h>
#include <libavutil/pixfmt.h>
}
static void SaveFrame(JNIEnv *pEnv, jobject pObj, char *image_name,jobject
pBitmap, int width, int height) {
    jmethodID sSaveFrameMID;
    jclassFFMPEGcls;
    //其中

saveFrameToPath是我们在

Java代码

FFMPEG类的一个方法，通过调用这个方法保存文件
```

```

        FFMPEGCls = pEnv->GetObjectClass(pObj);
        sSaveFrameMID = pEnv->GetMethodID( FFMPEGCls,
            "saveFrameToPath", "(Landroid/graphics/Bitmap;Ljava/lang/String;)V");
        jstring filePath = pEnv->NewStringUTF( image_name);
        pEnv->CallVoidMethod( pObj, sSaveFrameMID, pBitmap, filePath)
    }

```

```

static jobject createBitmap(JNIEnv *pEnv, int pWidth, int pHeight) {
    int i;
    //获取

```

Android系统

bitmap类和

createBitmap方法

```

    ID
    jclass javaBitmapClass = (jclass)(pEnv->FindClass( "android/graphics/Bitmap");
    jmethodID mid = pEnv->GetStaticMethodID(javaBitmapClass,
        "createBitmap", "(IILandroid/graphics/Bitmap$Config;)Landroid/graphics/Bitmap;");
    const wchar_t* configName = L"ARGB_8888";
    int len = wcslen(configName);
    jstring jConfigName;
    //wchar转换为

```

```

    jchar
    if (sizeof(wchar_t) != sizeof(jchar)) {
        jchar* str = (jchar*)malloc((len+1)*sizeof(jchar));
        for (i = 0; i < len; ++i) {
            str[i] = (jchar)configName[i];
        }
        str[len] = 0;
        jConfigName = pEnv->NewString( (const jchar*)str, len);
    } else {
        jConfigName = pEnv->NewString( (const jchar*)configName, len);
    }
    //调用

```

Android系统

bitmap类创建一个图片文件

```

        jclass bitmapConfigClass = pEnv->FindClass(
            "android/graphics/Bitmap$Config");
        jobject javaBitmapConfig = pEnv->CallStaticObjectMethod(bitmapConfigClass,
            pEnv->GetStaticMethodID( bitmapConfigClass, "valueOf",
            (Ljava/lang/String;)Landroid/graphics/Bitmap$Config;"), jConfigName);
        return pEnv->CallStaticObjectMethod( javaBitmapClass, mid,
            pWidth, pHeight, javaBitmapConfig);
    }

```

```

JNIEXPORT jboolean JNICALL Java_com_pref_video_FFMPEG_getFirstKeyFrame (JNIEnv *
env, jobject thisz, jstring video_name, jstring image_name)
{

```

```

    AVFormatContext *pFormatCtx = NULL;
    int
    videoStream;
    AVCodecContext *pCodecCtx = NULL;
    AVCodec
    *pCodec = NULL;
    AVFrame
    *pFrame = NULL;
    AVFrame
    *pFrameRGBA = NULL;
    AVPacket
    packet;

```

```

int                frameFinished;
jobjectbitmap;
void* buffer;
AVDictionary      *optionsDict = NULL;
struct SwsContext  *sws_ctx = NULL;
char *videoFileName,*imageFileName;
// 相关初始化

av_register_all();
//转换

jstring为

C++字符串

videoFileName = (char *)env->GetStringUTFChars(video_name, NULL);
imageFileName = (char *)env->GetStringUTFChars(image_name, NULL);
//打开视频文件

if(avformat_open_input(&pFormatCtx, videoFileName, NULL, NULL)!=0)
    return;
// 检索流信息

if(avformat_find_stream_info(pFormatCtx, NULL)<0)
    return;
av_dump_format(pFormatCtx, 0, videoFileName, 0);
// 查找第一个视频流

videoStream=-1;
for(int i=0; i<pFormatCtx->nb_streams; i++) {
    if(pFormatCtx->streams[i]->codec->codec_type==AVMEDIA_TYPE_VIDEO) {
        videoStream=i;
        break;
    }
}
if(videoStream==-1)
    return;
pCodecCtx=pFormatCtx->streams[videoStream]->codec;
//查找视频流的解码器

pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
if(pCodec==NULL) {
    fprintf(stderr, "Unsupported codec!\n");
    return;
}
// 打开解码器

if(avcodec_open2(pCodecCtx, pCodec, &optionsDict)<0)
    return;
// 申请一帧内存

pFrame=avcodec_alloc_frame();
pFrameRGBA=avcodec_alloc_frame();
if(pFrameRGBA==NULL)

```

```

        return;
    // 创建一个
    bitmap
    bitmap = createBitmap(env, pCodecCtx->width, pCodecCtx->height);
    if (AndroidBitmap_lockPixels(env, bitmap, &buffer) < 0)
        return;
    sws_ctx = sws_getContext(pCodecCtx->width, pCodecCtx->height, pCodecCtx->pix_fmt,
        pCodecCtx->width, pCodecCtx->height, AV_PIX_FMT_RGBA, SWS_BILINEAR,
        NULL, NULL, NULL);
    avpicture_fill((AVPicture *)pFrameRGBA, (uint8_t*) buffer, AV_PIX_FMT_RGBA,
        pCodecCtx->width, pCodecCtx->height);
    while(av_read_frame(pFormatCtx, &packet)>=0) {
        // 判断是否为视频帧

    if(packet.stream_index==videoStream) {
        // 解码视频帧

    avcodec_decode_video2(pCodecCtx, pFrame, &frameFinished, &packet);
    if(frameFinished) {
        // 从其原有格式转换为

    RGBA图像

        sws_scale (sws_ctx, (uint8_t const * const *)pFrame->data, pFrame-
        >linesize, 0,
            pCodecCtx->height, pFrameRGBA->data, pFrameRGBA->linesize);
        SaveFrame(env, thisz, imageFileName, bitmap,
            pCodecCtx->width, pCodecCtx->height);
        break;
    }
    }
    av_free_packet(&packet);
    }
    // 释放内存, 关闭解码器和文件

    AndroidBitmap_unlockPixels(env, bitmap);
    av_free(pFrameRGBA);
    av_free(pFrame);
    avcodec_close(pCodecCtx);
    avformat_close_input(&pFormatCtx);
    return true;
    }

```

代码清单8-8 Java类实现代码

```

public class FFMPEG {
    public String getFirstKeyFrame()
    {
        getFirstKeyFrame("视频文件

", "截取首帧图片文件

```

```
");
    if ((new File("截取首帧图片文件
")).exists())
    {
        return "截取首帧图片文件
";
    }
    return null;
}
//提供给
```

JNI保存的图片文件

```
private void saveFrameToPath(Bitmap bitmap, String pPath) {
    int BUFFER_SIZE = 1024 * 8;
    try {
        File file = new File(pPath);
        file.createNewFile();
        FileOutputStream fos = new FileOutputStream(file);
        final BufferedOutputStream bos = new BufferedOutputStream(fos, BUFFER_SIZE);
        bitmap.compress(CompressFormat.JPEG, 100, bos);
        bos.flush();
        bos.close();
        fos.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
//JNI 获取视频文件首帧图片文件接口

public native void getFirstKeyFrame(String vname,String iname);
//加载
```

FFMPEG相关的库

```
static {
    System.loadLibrary("avutil");
    System.loadLibrary("avcodec");
    System.loadLibrary("avformat");
    System.loadLibrary("swscale");
    System.loadLibrary("FFMPEG");
}
}
```

读到这里，如果读者对于代码实现的理解有一定的难度，或者觉得这种方式实现比较复杂的话，推荐读者直接使用编译好的FFMPEG

命令来实现，FFMPEG命令在Android命令行模式下执行结果如图8-20所示。

```
root@cancro:/data/local/tmp # ./ffmpeg -i /sdcard/1.mp4 -t 0 /sdcard/test.jpg
4 -t 0 /sdcard/test.jpg
ffmpeg version 2.3.1 Copyright (c) 2000-2014 the FFmpeg developers
  built on Aug  2 2014 22:39:12 with gcc 4.8 (GCC)
  configuration: --prefix='/home/magiclen/涓涓涓涓/ffmpeg-2.3.1/android/arm_neon' --enable-static --en
able-memalign-hack --disable-doc --disable-ffplay --disable-ffprobe --disable-ffserver --cross-prefi
x=/home/magiclen/android-ndk-r10/toolchains/arm-linux-androideabi-4.8/prebuilt/linux-x86_64/bin/arm-
linux-androideabi- --target-os=linux --arch=arm --enable-cross-compile --sysroot=/home/magiclen/andr
oid-ndk-r10/platforms/android-19/arch-arm/ --extra-cflags='-Os -fpic -marm -march=armv7-a -mcpu=neon
-mfloat-abi=softfp -mvectorize-with-neon-quad' --extra-ldflags='-Wl,--fix-cortex-a8'
```

图8-20 FFMPEG获取视频的首帧图片

在代码中，可以直接利用Java的“Runtime.getRuntime（）.exec”函数来实现。读者可以尝试一下，笔者在这里就不再赘述了。

7.比较图片

这部分代码主要是从批量截屏图片中找出和从视频源中提取的首帧原始图片相似的图片。需要读者注意的是，由于截屏的图片中可能会存在多张与基准图片相似的图片，所以要把基准图片按照文件名（文件名就是截屏系统时间）排序，第一次出现相似的图片就是测试需要找的首帧图片。关于图片相似度对比这里通过OPENCV SDK中比较直方图的方式来实现，具体实现如代码清单8-9所示。

代码清单8-9 比较图片相似度

```
//basePic是通过
FFMPEG截取的基准首帧图片文件路径
```



```

Mat base=Highgui.imread(basePic,1);
//把基准图像转化成

HSV图像

Imgproc.cvtColor(base, base, Imgproc.COLOR_RGB2HSV );
Mat bhist = new Mat();
MatOfInt histSize = new MatOfInt(25);
MatOfFloat ranges = new MatOfFloat(0f, 256f);
//计算图像的直方图

Imgproc.calcHist(Arrays.asList(base), new MatOfInt(0), new Mat(), bhist, histSize,
ranges);
//这里的

flist是批量截屏的文件列表，并且按截屏的时间前往后大排列

for (File f: flist)
{
Mat tmat=Highgui.imread(f.getPath(),1);
Imgproc.cvtColor(tmat, tmat, Imgproc.COLOR_RGB2HSV );
Mat thist = new Mat();
//计算图像的直方图

Imgproc.calcHist(Arrays.asList(tmat), new MatOfInt(0), new Mat(), thist, histSize,
ranges);
//比较查找图片和基准图片的相似度

double res = Imgproc.compareHist(bhist, thist, Imgproc.CV_COMP_CORREL);
//如果相似度大于

0.9,则认为这个就是我们要找的首帧图片

if (res >= 0.9)
{
return f.getName();
}
}

```

另外还需要说明的是，一些播放器在视频播放过程中，并非全屏播放。这里需要从截屏图片中提取视频播放区域的图片，具体方法可以参考UIAutomator一节，视频通过UIAutomator工具获取视频所在屏幕的起始位置和宽高，如图8-21所示。

根据UIAutomator获取起始坐标和结束坐标，然后通过Java类图片处理相关函数，把截屏图片的视频区域提取出来，这样再比较首帧图片相似度就不会受非全屏播放的影响了，关于这部分代码也不再赘述了，请读者自行实践。

8.计算响应时间

响应时间的计算相对来说比较简单，就是把找到的相似的图片的文件名利用Java的Long类的parseLong函数转换Long，然后再减去播放时间就是测试工具所需要的响应时间了，具体代码这里不再列出，请读者自行编写。

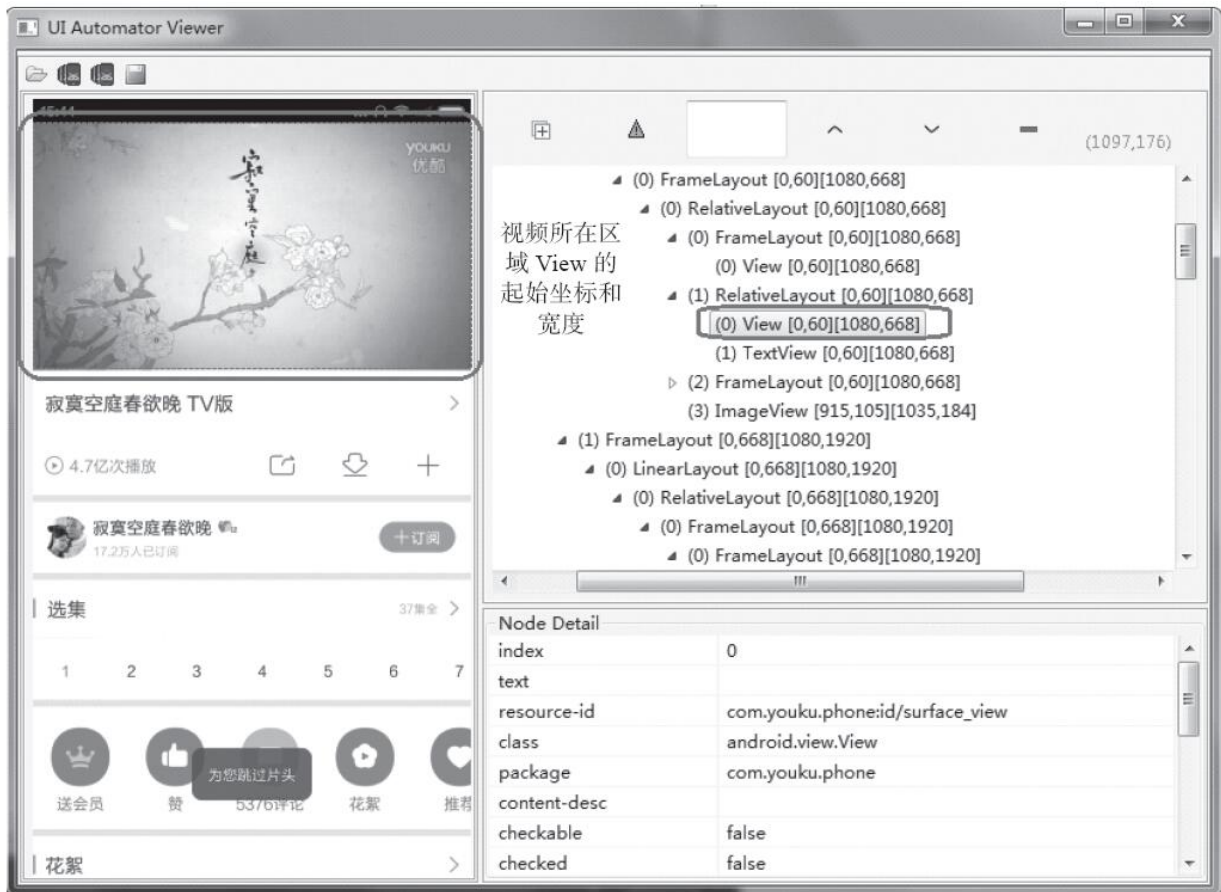


图8-21 UIAutomator获取视频区域位置信息

8.3.5 编译环境配置

到这里，已完成了整个工具的详细实现过程，为了确保工具代码正常编译和工具正常执行，需要对涉及的Android NDK和SDK两部分代码配置正确的权限与编译环境。

1.AndroidManifest.xml配置

大家都知道，在Android SDK开发中，如果需要访问资源和连接网络，必须在配置文件中对权限加以声明，否则将无法正常工作。而前面介绍的通过系统DownloadManger下载视频源，需要访问网络和读写存取卡，所以需要在配置文件中增加网络和存取卡的权限，配置如下面的代码所示。

```
<uses-permission android:name="android.permission.INTERNET" /> //访问网络权限

<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
//写存储卡权限
```

2.Android.mk配置

前面涉及input事件注入、批量截屏以及获取视频原始首帧的代码都是通过NDK方式实现的，为了能保证它们正常编译和链接，在配置

文件中需要配置头文件路径、链接库文件、编译的源文件以及产生的动态库或者可执行文件，具体相关配置如代码清单8-10所示。

代码清单8-10 Android.mk配置

```
#指明编译的工作路径

LOCAL_PATH:= $(call my-dir)
#配置利用

ffmpeg获取首帧编译环境

include $(CLEAR_VARS)
LOCAL_MODULE :=libavcodec
#注意，如果提示找不到对应文件的错误，需要根据

eclipse的提示信息调整相关路径

LOCAL_SRC_FILES :=$(LOCAL_PATH)/../FFMPEG/lib/libavcodec.so
include $(PREBUILT_SHARED_LIBRARY)
include $(CLEAR_VARS)
LOCAL_MODULE :=libavformat
LOCAL_SRC_FILES :=$(LOCAL_PATH)/../FFMPEG/lib/libavformat.so
include $(PREBUILT_SHARED_LIBRARY)
include $(CLEAR_VARS)
LOCAL_MODULE := libavutil
LOCAL_SRC_FILES :=$(LOCAL_PATH)/../FFMPEG/lib/libavutil.so
include $(PREBUILT_SHARED_LIBRARY)
include $(CLEAR_VARS)
LOCAL_MODULE := libswscale
LOCAL_SRC_FILES :=$(LOCAL_PATH)/../FFMPEG/lib/libswscale.so
include $(PREBUILT_SHARED_LIBRARY)
include $(CLEAR_VARS)
LOCAL_C_INCLUDES := $(LOCAL_PATH)
#编译所需要头文件和库文件

LOCAL_C_INCLUDES += $(LOCAL_PATH)/FFMPEG/include
LOCAL_LDLIBS := -llog -ljnigraphics -lz -landroid
#链接的动态库

LOCAL_SHARED_LIBRARIES := libavcodec libavformat libavutil libswscale
#指明需要编译的源文件

LOCAL_SRC_FILES :=FFMPEG.cpp
#指明编译产生的动态库文件

LOCAL_MODULE :=FFMPEG
```

#产生动态库文件，方便在

Java代码中进行加载

```
include$(BUILD_SHARED_LIBRARY)
```

#配置截屏编译环境

```
include $(CLEAR_VARS)
ANDROID_INCLUDE_PATH := $(LOCAL_PATH)/android/include
ANDROID_LIB_PATH := $(LOCAL_PATH)/android/lib
LOCAL_C_INCLUDES := $(LOCAL_PATH)
LOCAL_C_INCLUDES += \
    $(ANDROID_INCLUDE_PATH)/frameworks/base/native/include\
    $(ANDROID_INCLUDE_PATH)/frameworks/base/include\
    $(ANDROID_INCLUDE_PATH)/frameworks/base/libstagefright\
    $(ANDROID_INCLUDE_PATH)/frameworks/base/include/media/stagefright/openmax\
    $(ANDROID_INCLUDE_PATH)/frameworks/base/libstagefright/include \
    $(ANDROID_INCLUDE_PATH)/frameworks/native/include \
    $(ANDROID_INCLUDE_PATH)/hardware/libhardware/include \
    $(ANDROID_INCLUDE_PATH)/system/core/include \
    $(ANDROID_INCLUDE_PATH)/external/skia/include/core \
    $(ANDROID_INCLUDE_PATH)/external/skia/include/effects \
    $(ANDROID_INCLUDE_PATH)/external/skia/include/images \
    $(ANDROID_INCLUDE_PATH)/external/skia/src/ports \
    $(ANDROID_INCLUDE_PATH)/external/skia/include/utils
LOCAL_LDLIBS := -llog -ljnigraphics -lz -landroid
LOCAL_LDLIBS += -Wl,-rpath=$(ANDROID_LIB_PATH)\
    -L$(ANDROID_LIB_PATH)\
    -lutils\
    -lbinder\
    -lskia\
    -lui \
    -lgui
LOCAL_CFLAGS += -DHAVE_SYS_UIO_H #redefinition of 'struct iovec'
```

#编译

takeshot和

event两个

cpp文件

```
LOCAL_SRC_FILES :=takeshot.cpp \
    event.cpp
LOCAL_MODULE :=takeshot
```

#编译可执行文件

```
include$(BUILD_EXECUTABLE)
```

#配置

OPENCV编译环境

```
OPENCV_CAMERA_MODULES:=on
OPENCV_INSTALL_MODULES:=on
```

#指明

opencv.mk文件所在位置，这个主要取决于下载解压文件所在的路径

```
include F:/opensource/library/OPENCV-2.4.10/sdk/native/jni/OPENCV.mk
```

8.3.6 工具安装

工具编译成功后，会生成takeshot可执行文件和prefvideo.apk安装文件，而prefvideo.apk在运行过程中快速截屏和模拟点击播放按钮都是通过takeshot命令来实现的。所以这里需要把这个命令复制到手机系统/system/bin目录下，并改为可执行文件。apk的安装和其他应用程序安装相同，这里就不做介绍了。下面是可执行文件的安装过程，具体操作如下：

(1) 命令takeshot位于perfvideo工程所在目录libs\armeabi下，首先通过adb命令push到手机的SD存储卡上，具体如图8-22所示。

```
E:\project\java\prefVideo\libs\armeabi>adb push takeshot /sdcard/  
1078 KB/s (17668 bytes in 0.016s)
```

图8-22 文件复制到SD卡目录

(2) 更改/system/bin目录权限，如图8-23所示。

```
E:\project\java\prefVideo\libs\armeabi>adb shell  
esshell@cancro:/ $ u  
su  
root@cancro:/ # cd /system/bin  
cd /system/bin  
root@cancro:/system/bin # mount -o rw,remount /system  
mount -o rw,remount /system
```

图8-23 文件从SD卡复制到系统目录

(3) 把两个命令复制到/system/bin目录并更改为可执行文件，如图8-24所示。

```
127!root@cancro:/system/bin # cp /sdcard/takeshot .  
cp /sdcard/takeshot .  
root@cancro:/system/bin # chmod 777 takeshot  
chmod 777 takeshot
```

图8-24 更改文件为可执行文件

8.4 方案优缺点

前面介绍了相关视频首帧响应时间自动化测试方案，下面总结一下本方案的优缺点。

方案优点:

- 独立在手机中运行，不需要其他辅助设备。
- 能够自动且精准地计算出首帧响应时间。

方案缺点:

- 屏幕截屏和注入input事件需要手机ROOT。
- 测试机型配置一般需要4核及以上和2G内存及以上的内存。（主要原因是机型配置太低，快速截屏可能会影响测试结果的精准度）

8.5 本章小结

本章案例侧重于描述视频首帧响应时间工具的设计，以Androidos4.4系统下QQ浏览器视频首帧响应时间为例，详细、完整地介绍了整个方案设计思路，并对设计思路中的关键点做出了详细分析和代码实践，对于读者可能会遇到的共性问题也做了详细解释。另外，对于一些关键点设计思路可能存在多种实现方案，在本章也一一列出，但对于细节没有逐一展开，所以这部分内容需要读者自行练习。

本章内容比较丰富，涉及的知识点也多。通过本章阅读，相信读者一定对于OPENCV图像识别技术、FFMPEG视频解码技术以及Android系统相关源码有了进一步的了解和认识。此方案不仅适合浏览器网页视频播放测试，也适用于Android系统下任何App视频播放器响应时间的性能测试，只需要针对自己的产品稍做变形，就可以实现对应的测试方案。

第9章 应用宝BVT测试案例

BVT (Build Verification Test) 测试指通过自动化手段，验证新生成的软件版本在功能上是否完整、主要的软件特性是否正确。在项目周期短、版本快速迭代的情况下，**BVT**测试可以快速完成对新版本的验证工作，避免低质量的版本落入测试人员手中，浪费人力；另外，日常的**BVT**测试还可以起到监控的作用，有助于把控整体的质量风险。此外，应用宝项目组采用**FT (Feature Team)** 模式，整个项目组分为多个**FT**，而每个**FT**又同时有多个需求分支在并行运作着，几乎每天都有新特性合入主干，因此很有必要将分支合流前**rebase**测试、合流后主干验证测试等环节加入**BVT**自动化测试，以持续验证新特性未破坏原有系统。

本章将从测试工程概览、测试用例编写、测试报告生成、跨应用处理、测试覆盖率度量等维度介绍**BVT**自动化测试在应用宝中的实际应用情况。第一小节介绍基于**Robotium**的测试工程概览，了解**Android**端自动化测试的动作模式。第二小节介绍基于**Robotium**测试用例的编写，包括如何使用例更加健壮稳定、降低维护成本等。第三小节介绍结合**Spoon**测试报告生成，以及如何更好地进行出错重试与截图。第四小节介绍结合**UIAutomator2.0**版本进行跨应用测试。最后介绍测试覆盖率的度量以及测试效益的思考。本章知识结构图如图9-1所示。

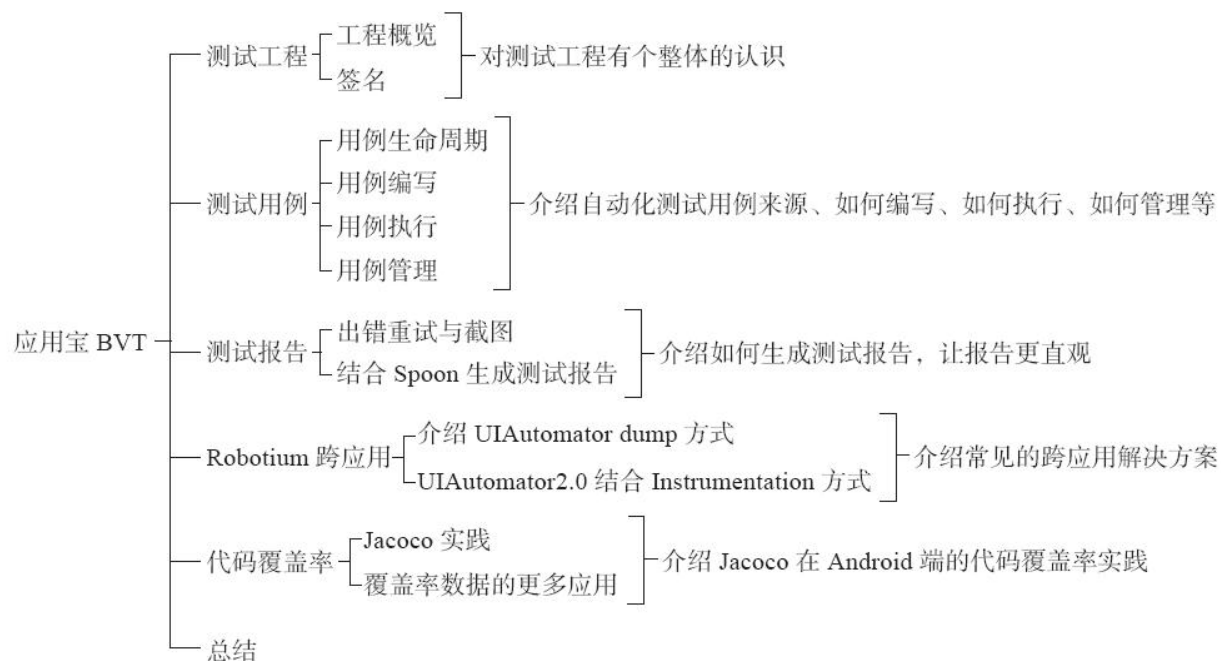


图9-1 本章知识结构图

9.1 测试工程

9.1.1 测试工程概览

使用Robotium进行自动化测试，测试工程为一个Android Junit Test工程，可以依赖被测工程，也可以选择独立存在。如果需要引用被测工程中的源码，则需要在测试工程的Build Path中添加对被测工程的引用，如图9-2所示。

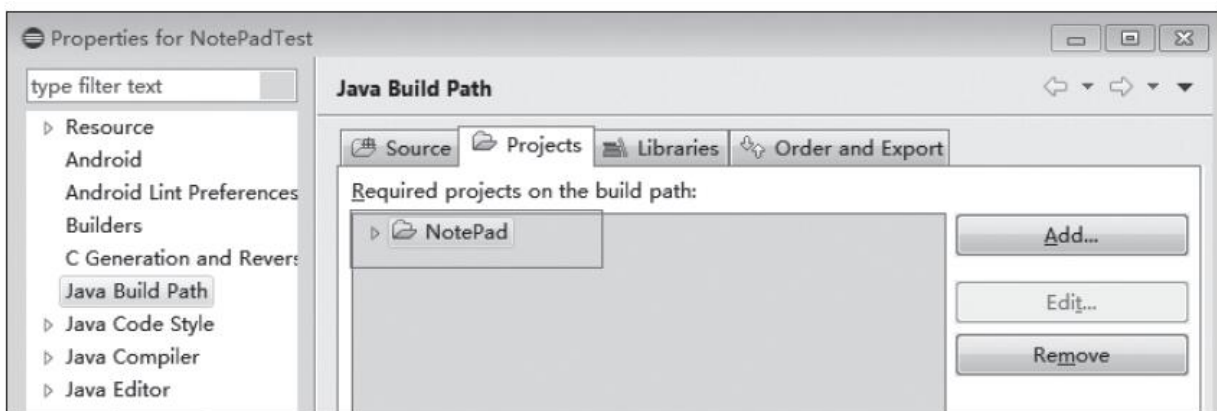


图9-2 在测试工程中引用被测工程

关联被测工程源码的好处在于可以调用被测工程的代码，因此可以更容易地获取被测应用内部的状态，例如拿到被测应用ListView内部填充的数据等。而这样也会带来一些弊端：

·测试工程的自动化编译打包也需要关联被测工程，脚本复杂度及维护成本增加。

·如果采用**R.id.xxx**方式获取控件的话，被测工程增加、删除布局文件都可能影响测试工程的编译结果。

·如果被测应用进行了代码混淆，引用被测工程的代码复杂度将大大提高。

鉴于此，应用宝采用的是脱离被测工程的方式，同一份测试**APK**可以同时测试多个版本的被测应用。另外，即使大家选择有源码的方式，也不建议使用**R.id.xxx**的方式获取控件。

然后，测试工程引用**Robotium**相应版本的**jar**包即可，需要注意的是**Robotium**这个**jar**中的**class**类是**Android**手机中未存在的，因此在**Order and Export**中需要勾选上。测试工程其余整个结构与**Android**工程一样，可以有自己的布局文件、资源文件等。此外，测试工程需要在**AndroidManifest.xml**文件中注册**Instrumentation**用于指定被测应用，如下面的代码所示。

```
<instrumentation
    android:targetPackage="com.robotium.android.notepad"
    android:name="android.test.InstrumentationTestRunner" />
```

在同一个测试工程中我们可以只注册一个Instrumentation，也可以同时注册多个，例如当被测应用有多个，而测试工程又不想分别建立时，则可以使用注册多个的方法。首先，编写一个继承自android.test.InstrumentationTestRunner的自定义InstrumentationTestRunner，然后同样地在AndroidManifest.xml中注册，如下面的代码所示。

```
<instrumentation
    android:targetPackage="com.robotium.android.anothernotepad"
    android:name=".instrumentation.InstrumentationTestRunner" />
```

9.2.1 测试工程签名

由于测试用例的代码在运行过程中是以**Instrumentation**注入的方式，和被测应用在同一进程，因此需要测试工程与被测工程签名一致。要使两个工程签名一致有两种方式，一种是重签名被测工程；另一种则是不改变被测工程，而直接使用被测工程的**Key**签名测试工程。

1.重签名被测App

当进行第三方竞品测试，无法获取到第三方App的签名**Key**时，可以使用重签名的方式。以使用**Android**中的**debug.keystore**进行重签名为例。

- (1) 解压被测应用的**APK**文件。
- (2) 删除其中的签名文件**META-INF**目录。
- (3) 重压缩，并重命名回**APK**文件。

以上步骤主要目的是删除原来**APK**文件里的签名文件**META-INF**目录，即去掉原有签名，然后在命令行中调用**JDK**中的**jarsigner**工具进行签名即可，代码如下：

```
> jarsigner -verbose -keystore ~/.android/debug.keystore -storepass android  
-keypass android -signedjar applicationName_resigned.apk applicationName.apk
```

在JDK7及以上环境中则还需要增加sigalg和digestalg参数，代码如下：

```
> jarsigner -verbose -keystore ~/.android/debug.keystore -storepass android  
-keypass android sigalg MD5withRSA digestalg SHA1 -signedjar applicationName_  
resigned.apk applicationName.apk androiddebugkey
```

这里的debug.keystore位于C: \Users相应用户名下的.android目录中，测试工程在Eclipse中默认使用的也是这个debug.keystore，因此被测工程也使用debug.keystore签名后，两者的签名就一致了。

2.签名测试App

在实际项目中，如果是自己的项目，显然是不希望对被测App进行重签名的，原因如下：

- 每日进行测试的包众多，一一进行重签名影响效率。
- 如微信、应用宝等应用做了签名防护措施，重签名后将导致应用部分功能不可用甚至直接无法启动。

因此，自己的项目可以拿到被测工程的签名Key，使用该Key对测试工程进行签名即可，Eclipse中默认编译出来的测试APK使用的是

debug.keystore，可以在“Preferences—Android—Build”中设置为使用被测工程的keystore，如图9-3所示。

这样即可在不改变被测应用的情况下，对被测应用进行测试。此外，我们肯定不希望自动化测试借用Eclipse进行编译、打包、签名，而是希望测试代码也可以持续集成起来，因此有必要使用构建工具如ant进行编译、打包、签名。由于测试工程结构与普通的Android项目一致，网上资料也较多，因此测试工程使用ant进行编译、打包、签名在这就不再赘述了。

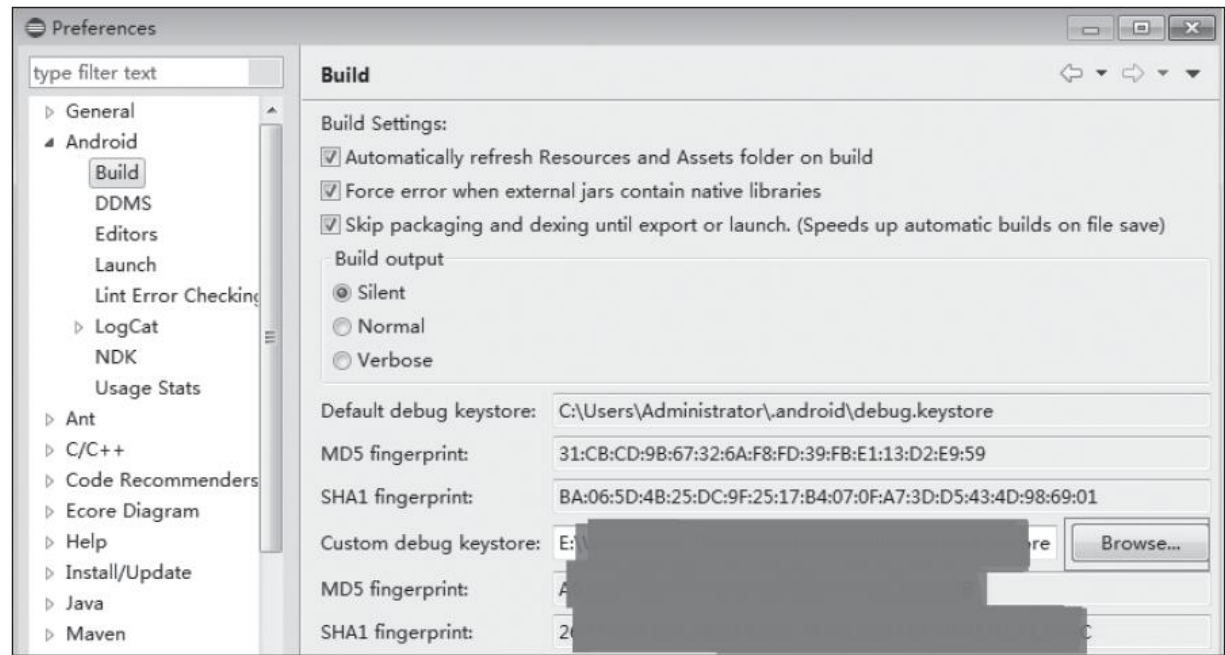


图9-3 测试工程使用自定义的keystore

9.2 测试用例

9.2.1 测试用例生命周期

测试用例基于Android Junit，每个用例遵循下面三个步骤。

- (1) 执行setUp () 方法，用于初始化。
- (2) 执行以public且方法名以test开头的用例方法。
- (3) 执行tearDown () 方法，用于释放资源等。

如代码清单9-1所示，用例先对构造函数进行初始化，通过反射机制获取了要启动的Activity类名，然后开始进入用例的生命周期。先执行setUp () 方法，并实例化Solo对象，new Solo (getInstrumentation () , getActivity ()) 中的第二个参数为调用的基类ActivityInstrumentationTestCase2中的getActivity () 方法，如果指定的Activity未启动，将通过Intent唤起该Activity。随后，开始执行testAddNote () 方法，进行实际的自动化测试。最后，调用tearDown () 方法，solo.finishOpenedActivities () 将关闭所有已打开的Activity，完成一个完整的用例测试过程。

代码清单9-1 基于apk测试示例

```

package com.robotium.test;
import android.test.ActivityInstrumentationTestCase2;
import com.robotium.solo.Solo;
public class NotePadTest extends ActivityInstrumentationTestCase2{
    private static final String LAUNCHER_ACTIVITY_FULL_CLASSNAME =
"com.robotium.android.notepad.NotesList";
    private static Class launcherActivityClass;
    static{
        try
        {
            launcherActivityClass=Class.forName(LAUNCHER_ACTIVITY_FULL_CLASSNAME);
        } catch (ClassNotFoundException e){
            throw new RuntimeException(e);
        }
    }
    private static final String TAG = NotePadTest.class.getSimpleName();
    private Solo solo;
    public NotePadTest() {
        super(launcherActivityClass);
    }
    @Override
    public void setUp() throws Exception {
        //setUp() is run before a test case is started.
        //This is where the solo object is created.
        super.setUp();
        solo = new Solo(getInstrumentation(), getActivity());
    }
    @Override
    public void tearDown() throws Exception {
        //tearDown() is run after a test case has finished.
        //finishOpenedActivities() will finish all the activities that have been
opened during the test execution.
        solo.finishOpenedActivities();
        solo = null;
        super.tearDown();
    }
    public void testAddNote() throws Exception {
        //Unlock the lock screen
        solo.unlockScreen();
        solo.clickOnMenuItem("Add note");
        //Assert that NoteEditor activity is opened
        solo.assertCurrentActivity("Expected NoteEditor activity", "NoteEditor");
        //In text field 0, enter Note 1
        solo.enterText(0, "Note 1");
        solo.goBack();
        //Clicks on menu item
        solo.clickOnMenuItem("Add note");
        //In text field 0, type Note 2
        solo.typeText(0, "Note 2");
        //Go back to first activity
        solo.goBack();
        //Takes a screenshot and saves it in "/sdcard/Robotium-Screenshots/".
        solo.takeScreenshot();
        boolean notesFound = solo.searchText("Note 1") && solo.searchText("Note
2");
        //Assert that Note 1 & Note 2 are found
        assertTrue("Note 1 and/or Note 2 are not found", notesFound);
    }
}

```

9.2.2 测试用例编写

测试用例编写的质量直接关系到用例的稳定性、维护成本以及是否能发现有效问题等，因此是自动化测试中的关键一环。

首先，确定测试用例的来源。

当开始准备编写自动化测试用例时，需要确定测试用例的来源，即需要明确以下几个方面：

- 哪些功能是主要功能，哪些功能可以自动化。
- 用例的优先级、作用的测试阶段。
- 测试一个功能需要哪些验证点。

不同的项目组需要思考的点可能不一样，但目的是一致的，需要明确测试用例的来源，而不是任意地开始编写用例。应用宝中采用 **CheckList** 的形式，通过与各业务线讨论评审的方式确定关键功能、是否自动化、用例优先级、测试验证点等，如图9-4所示。

ID	用例名称	优先级
76756390	CASE 插件动态下发-更新	P1
76756388	CASE 插件动态下发-下载	P0
76754205	CASE 搜索直达区内容外显	
76736565	CASE 小米攻防卡	P0
76736563	CASE 管理-辅助功能	P0

图9-4 确认测试用例来源

明确了测试用例的来源后，一个用例的步骤及验证点就基本确定了。

其次，应该合理地去设计自动化测试用例。

接下来本小节将通过代码清单9-2中的示例从工程角度及用例设计角度来介绍BVT自动化测试用例的编写。

代码清单9-2 BVT测试用例示例

```
/**
 * 测试从下载管理页中下载安装应用
</br></br>
 *
 * 1. 查看下载管理页中是否有下载任务，没有的话则从发现频道中添加一个
</br>
 * 2. 进入下载任务管理页，点击下载任务中的“继续”按钮，继续下载并安装应用
</br>
 * 3. 断言应用是否安装成功；断言安装成功后，下载按钮状态是否变为“打开”
</br>
 * 4. 为确保后面的测试，会卸载该应用
</br>
 */
public void test76410589_Download_FromTaskList_ShouldInstallSucessful(){
```

```

operation.deleteAllDownloadTasks();
//若测试前下载任务为空，则先添加一个

if(operation.isDownloadPageEmpty()){
    operation.addDownloadTaskFromTab();
    operation.sleepWait();
}
operation.enterDownloadActivity();
DownloadTaskListItem downloadTaskListItem =
operation.getDownloadTaskListItemByIndex(0);
TextView downloadButton = downloadTaskListItem.getDownloadButton();
appName = downloadTaskListItem.getAppName().getText().toString();
if(!downloadButton.getText().toString().equals(YYBConstant.STAT_OPEN)
&& !downloadButton.getText().toString().equals(YYBConstant.STAT_PAUSE)){
    solo.clickOnView(downloadButton);
    operation.sleepWait();
}
operation.waitForTextStat(downloadButton, YYBConstant.STAT_INSTALLING,
TimeUtils.THREE_MINI);
operation.waitForTextStat(downloadButton, YYBConstant.STAT_OPEN,
TimeUtils.ONE_MINI);
operation.sleepWait();
LogUtils.logD(TAG, "isAppInstalled after install:" +
operation.isAppInstalled(appName));
assertTrue(appName + "应该要安装到手机上了

", operation.isAppInstalled(appName));
operation.assertDownloadBtnStat(downloadButton, YYBConstant.STAT_OPEN);
operation.sleepWait();
operation.uninstallByAppName(appName);
operation.goBackToMainActivity();
}

```

在设计自动化测试用例时，除了实现用例来源中的功能步骤外，用例的原子性是需要特别注意的，这将影响到多个用例在一起时是否可以高效稳定地运行。用例的原子性，即用例间应该保持相对独立，不因用例执行的先后顺序而彼此干扰。如代码清单9-2所示，**deleteAllDownloadTasks**（）方法用于初始化环境，清空原有的管理页中的任务列表，避免先前执行用例对该用例造成影响。然后常规地实现了用例来源中的步骤功能，自动添加了一个下载任务，进入下载管理页，下载并安装应用。最后，为了保持用例的原子性，执行完该

用例后不影响接下来的用例，因此调用`uninstallByAppName`（`appName`）方法将刚才安装的应用卸载。

再次，应该以工程的视角去看待测试用例。

测试代码也应该以工程的视角去看待，包括配置管理、结构管理、项目化运作等。在编写测试用例过程中也应该尽可能地从工程角度在代码易用性、维护性方面多加考虑。

1. 依赖与耦合

耦合度就是某模块（类）与其他模块（类）之间的关联、感知和依赖的程度，是衡量代码独立性的一个指标。在耦合度很高的情况下，维护代码时修改一个地方就会牵连到很多地方；而相反地，如果耦合度很低，往往又意味着重复代码多，如同一盘散沙，难以维护。因此，需要平衡依赖与耦合之间的关系，让代码稳定、健壮且易于维护。

在测试代码中，常常有一些步骤是许多用例都需要执行的，那么就应该将这些通用的步骤提取出来，作为公共的方法，且这个方法需要确保有足够的稳定性，因为它的健壮与否，关系到多个用例的健壮性。如代码清单9-3所示，除了`solo`常规的操作方法外，在业务上将常用的一些跳转步骤、环境构造、业务层的断言等进行封装。统一封装了进入下载管理页的方法，每个用例在需要进入该页面时调用

enterDownloadActivity () 方法即可，而如果跳转至该页面的UI有变动，则仅需修改该方法。此外，代码中通过ViewId常量类来统一管理控件ID，避免UI变动时到处修改测试用例。

从依赖与耦合的角度看，但凡测试代码中有重复的地方，均应该将其提取封装，并确保封装方法的稳定性，Selenium中使用的PageObject模式也是基于这样的目的的。

代码清单9-3 测试用例中对跳转统一封装

```
/**
 * 下载管理
 */
</br>
 * @return
 */
public boolean enterDownloadActivity(){
    checkIsMainActivity();
    RelativeLayout mBtnDownload =
(RelativeLayout)solo.getView(ViewId.mybtndownload);
    solo.clickOnView(mBtnDownload);
    sleeper.sleepWait();
    return isExpectedActivity(YYBConstant.DOWNLOAD_ACTIVITY);
}
```

2.面向对象的控件获取方式

由于应用宝中的控件很多是以列表Item形式存在的，在每个Item中，应用名、应用大小、下载按钮等的控件ID在不同的界面中均是相同的，且也不希望用例代码中包含大量ID名。因此，BVT用例在获取控件时采用的是面向对象的方式，如代码清单9-4所示，将列表中一个

Child中的多个控件封装成一个SimpleApp对象，测试用例中就不再需要关心控件的细节了，加快了用例的编写速度并降低了维护成本。

代码清单9-4 测试用例面向对象的获取控件方式

```
public SimpleApp getSimpleApp(RelativeLayout relativeLayout){
    SimpleApp simpleApp = new SimpleApp();
    simpleApp = pieceSimpleApp(relativeLayout);
    simpleApp.setAppTitle((TextView)
relativeLayout.findViewById(holo.getId(ViewId.title)));
    if(simpleApp.getAppTitle() == null){
        LogUtils.logD(TAG, "title is special");
        simpleApp.setAppTitle((TextView)
relativeLayout.findViewById(holo.getId(ViewId.app_name_txt)));
    }
    return simpleApp;
}
```

3.其他

当然，测试代码也应该有代码规范，包含命名规范、编写规范、注释规范等，以使测试用例能高效、有质量地运转起来。

最后，应该验证测试用例的有效性。

自动化测试用例本身也是需要经过验证与测试的，一个测试用例本身运行通过了并不一定代表用例就是有效的。例如可能因为检查点判断有问题导致该用例始终通过，而一般当用例开始交付运行后，如果一直是通过的，那么往往就不会再有人关注，且测试人员会认为该模块已经有自动化测试去保障，从而容易忽略基本的测试，所以往往无效的自动化测试用例比没有自动化测试的测试用例更可怕，需要警

惕出现无效的测试用例。在编写测试用例时需要验证用例的有效性，在测试用例交付使用后，也应该定期地关注测试用例的运行情况及其有效性。

9.2.3 测试用例执行

以Eclipse为例，运行测试用例时选择需要执行的用例类，若已配置Run Configuration，则右击Run As选择Android Junit Test即可。若还未配置Run Configuration，则右击Run As——Run Configuration进行配置，可以设定要执行的类名及Instrumentation runner等，如图9-5所示。

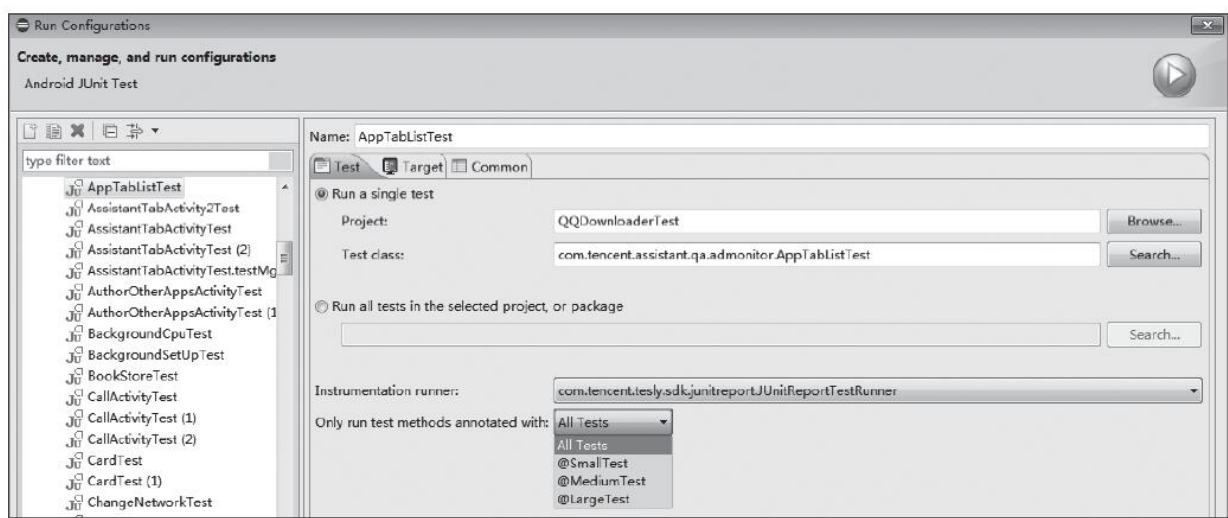


图9-5 Run Configuration配置

除了在IDE中运行测试用例外，也可以使用adb命令行方式运行测试用例。基于Instrumentaion的测试用例可以使用android.test包下的InstrumentationTestRunner驱动执行。命令行方式为：

```
$ adb shell am instrument -w <test_package_name>/<runner_class>
```

主要参数说明见表9-1。

表9-1 主要参数说明

参数	值	说明
<test_package>	测试工程的包名	在测试工程的 AndroidManifest 中定义的自身包名
<runner_class>	使用的 test runner 类名	例如 Android、Test、InstrumentationTestRunner

am instrument的标志位说明见表9-2。

表9-2 am instrument的标志位说明

标志	值	说明
-w	无	强制 am instrumentation 等待直至 Instrumentation 运行完成，这个标志位的作用在于，如果不使用该选项，则控制台未等用例执行结束即已退出，那样不方便从控制台中看到测试结果
-r	无	输出执行过程中的原始信息，该标志位是为性能测量而设计的，常与 -e perf true 参数同时使用
-e	<test_options>	提供键值对形式的参数选项，可提供多个参数选项

am instrument的主要参数选项见表9-3。

表9-3 am instrument的主要参数选项

键	值	说明
package	<Java_package_name>	指定要执行的包
class	<class_name>	指定要执行的类
	<class_name>#method name	指定要执行的某个类中的方法
size	[small medium large]	执行的带有指定注解的用例，注解可以是 @SmallTest、@MediumTest 或 @LargeTest
debug	true	设置是否以 debug 模式运行用例
perf	true	设置是否运行实现了 PerformanceTestCase 的用例，当使用该选项时，一般也同时开启了 -r 标志位，以输出原始信息
EMMA	true	设置是否开启使用 EMMA 收集代码覆盖率，EMMA 输出的默认路径为 /data/<app_package>/coverage.ec，也可以通过 coverageFile 选项指定路径。需要注意的是，开启该选项的前提是，被测应用构建打包时需要为 EMMA-instrumented 的包
coverageFile	<filename>	指定 EMMA 输出的路径

通过`am instrument`的各参数选项的组合，可以灵活指定要执行的测试用例，例如：

执行单个类里的某个指定用例代码如下：

```
$ adb shell am instrument -w -e class com.android.foo.FooTest#testFoo com.android.foo/android.test.InstrumentationTestRunner
```

执行多个类中的所有用例代码如下：

```
$ adb shell am instrument -w -e class com.android.foo.FooTest,com.android.foo.TooTest com.android.foo/android.test.InstrumentationTestRunner
```

编译打包时使用**EMMA**进行插桩，当需要自动化用例收集代码覆盖率时，可以使用如下命令开启：

```
$ adb shell am instrument -w -e coverage true coverageFile /sdcard/myFile.ec class com.android.foo.FooTest,com.android.foo.TooTest com.android.foo/android.test.InstrumentationTestRunner
```

将**coverage**设置为**true**后，运行完自动化用例，将会在**/sdcard**目录下生成**myFile.ec**文件，再通过**EMMA**工具可以生成覆盖率报告，代码覆盖率相关知识可以参见9.5节，将有更详细的介绍。

9.2.4 测试用例管理

当编写了较多测试用例时，就需要将测试用例进行分类管理，以方便统一维护及用例分级。基于Junit的测试可以使用TestSuite的方式进行管理，如代码清单9-5所示。

代码清单9-5 使用TestSuite的方式进行用例管理

```
public class ExampleTestSuite extends TestSuite{
    public static Test suite() {
        TestSuite tsSuite = new TestSuite();
        tsSuite.addTestSuite(GameTabActivityTest.class);
        tsSuite.addTest(TestSuite.createTest(AppTabActivityTest.class,
            "testFoundTab_CheckQuickEntry"));
        return tsSuite;
    }
}
```

由于在测试执行时，不同的用例执行时间长短不同，且作用的测试阶段也各不相同，因此在进行用例管理时，需要明确用例的级别，例如区分是核心功能用例还是普通用例，从而将不同级别的用户放于一处进行管理，在执行时才可以有针对性地进行测试。

9.3 测试报告

9.3.1 Spoon介绍

Spoon是一个主导有okhttp、retrofit、leakcanary等众多优秀开源项目的Square公司在GitHub上的开源项目，致力于改善基于Instrumentation的测试。通过分布式地在多部手机上同时执行基于Instrumentation的测试用例，并且在测试完成后生成统一的拥有测试结果概览、截图、运行时日志等功能的HTML形式测试报告，Spoon可以更加快速、有效地对Android终端进行自动化测试。

项目开源地址：<https://github.com/square/spoon>。

测试报告可以方便地看到哪个用例在哪部手机上执行失败，结果概览如图9-6所示。

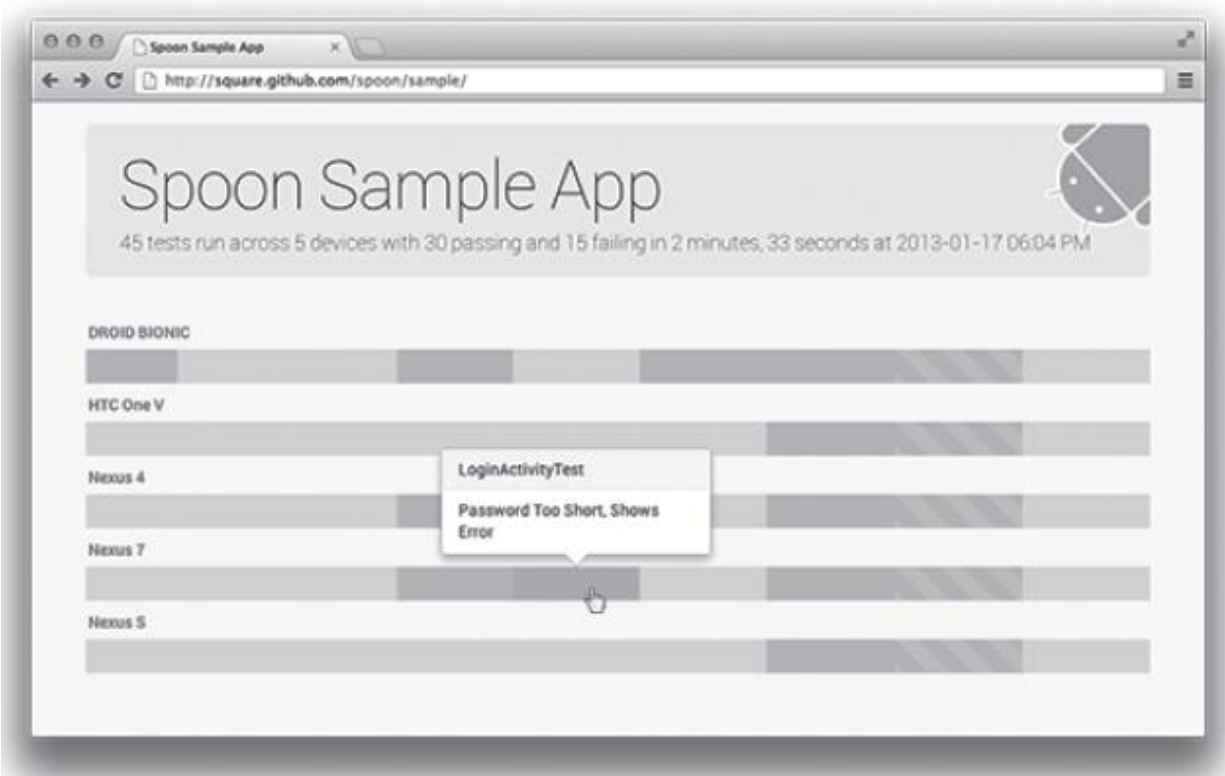


图9-6 多机同时执行后的测试结果概览

测试报告包含执行过程中的截图，如图9-7所示。



图9-7 用例执行过程中的截图

测试报告也包含每个用例的运行日志，如图9-8所示。

Another Long Name Because It Is Humorous And Testing Things Like This Is Important			
Test errorred in 0 seconds on Droid			
Timestamp	Level	Tag	Message
02-12-10:28:19.560	info	TestRunner	started: testAnotherLongNameBecauseItIsHumorousAndTestingThingsLikeThisIsImportant(com.example.spoon.ordering.tests.MiscellaneousTest)
02-12-10:28:19.623	debug	dalvikvm	GC_EXPLICIT freed 936 objects / 124856 bytes in 62ms
02-12-10:28:19.701	debug	dalvikvm	GC_EXPLICIT freed 14 objects / 688 bytes in 70ms
02-12-10:28:19.701	info	TestRunner	failed: testAnotherLongNameBecauseItIsHumorousAndTestingThingsLikeThisIsImportant(com.example.spoon.ordering.tests.MiscellaneousTest)
02-12-10:28:19.701	info	TestRunner	----- begin exception -----
02-12-10:28:19.716	info	TestRunner	java.lang.IllegalStateException: Explicitly testing unexpected, nested exceptions.
02-12-10:28:19.716	info	TestRunner	at com.example.spoon.ordering.tests.MiscellaneousTest.testAnotherLongNameBecauseItIsHumorousAndTestingThingsLikeThisIsImportant(MiscellaneousTest.java:106)
02-12-10:28:19.716	info	TestRunner	at java.lang.reflect.Method.invokeNative(Native Method)
02-12-10:28:19.716	info	TestRunner	at java.lang.reflect.Method.invoke(Method.java:521)
02-12-10:28:19.716	info	TestRunner	at android.test.InstrumentationTestCase.runMethod(InstrumentationTestCase.java:204)
02-12-10:28:19.716	info	TestRunner	at android.test.InstrumentationTestCase.runTest(InstrumentationTestCase.java:194)
02-12-10:28:19.716	info	TestRunner	at junit.framework.TestCase.runBare(TestCase.java:127)
02-12-10:28:19.716	info	TestRunner	at junit.framework.TestResult\$1.protect(TestResult.java:106)
02-12-10:28:19.716	info	TestRunner	at junit.framework.TestResult.runProtected(TestResult.java:124)
02-12-10:28:19.716	info	TestRunner	at junit.framework.TestResult.run(TestResult.java:109)

图9-8 用例的运行日志

通过Spoon来执行基于Instrumentation的测试用例，可以很方便地生成富于展示的测试报告，通过报告中运行时截图及运行时日志可以快速定位用例失败的原因。此外，在报告中的junit-reports目录下，还生成有基于Junit格式的xml报告，可以通过解析该目录下的xml报告获得详细的包括用例执行总数、用例通过总数、用例失败总数、用例执行时间等数据，以方便地进行测试数据统计。

通过上文的介绍可知，基于Instrumentation的测试用例都可以通过Spoon执行并生成测试报告，而Robotium框架编写的测试用例就是基于

Instrumentation的，因此两者可以很好地进行结合以达到增强测试报告精准度的目的。

测试报告通过调用Spoon Runner的jar包执行测试后生成，使用方法为java-jar命令行方式，Spoon Runner的主要参数如代码清单9-6所示。

代码清单9-6 Spoon Runner的主要参数

Options:	
--apk	被测
APK包所在的路径	
--fail-on-failure	当出现
failure时，发现非	
0的退出码	
--output	测试报告的输出路径，默认为
spoon-output	
--sdk	Android SDK的路径，若已配置可不填
--test-apk	测试
APK的路径	
--title	测试报告显示的标题
--class-name	测试用例类名，需要为带包名的全称
--method-name	测试用例方法名
--no-animations	禁止进行截图的
gif生成	
--size	只运行包含相应注解的用例
(small、	

```
medium、  
large)  
--adb-timeout          设置每个用例支持的超时时间（默认为  
10分钟）
```

在执行时需要指定被测的APK及测试APK所在的路径，还需要指定测试用例名等参数。在执行时，Spoon会自动将被测APK与测试APK安装至手机，代码如下（示例中使用的%apkpath%为参数，实践中需要替换为实际的路径）：

```
java -Dencoding=UTF-8 -Dfile.encoding=UTF-8 -jar spoon-runner-1.1.3-SNAPSHOT-  
jar-with-dependencies.jar --title "YYB Continuous Integration Regression Test"  
--apk %apkpath% --test-apk %testapk_path% --class-name %test_class%
```

测试完成后将在spoon-output目录下生成如图9-9所示的目录结构，此报告为HTML形式的静态报告，通过Web服务器即可对外提供访问。



图9-9 spoon-output报告的目录结构

9.3.2 结合Spoon的出错重试与截图

在编写用例过程中，会有以下需求点：

- 用例失败时可以即时重试，避免过多的误报情况。
- 用例失败时可以包含执行过程中的截图，便于根据当时场景定位问题。

先来看看失败重试的实现，测试基类`InstrumentationTestCase`继承自Junit中的`TestCase`，采用的也是通过`runTest ()`方法适配各个Test的模式。在`runTest ()`中，规定了测试用例需要为`public`的`Method`。此外，通过`@FlakyTest (tolerance=3)`注解的形式，当测试用例失败时会进行重试，重试次数由`tolerance`的值指定。如代码清单9-7所示。

代码清单9-7 `InstrumentationTestCase`中的`runTest ()`源码实现

```
/**
 * Runs the current unit test. If the unit test is annotated with
 * {@link android.test.UiThreadTest}, the test is run on the UI thread.
 */
@Override
protected void runTest() throws Throwable {
    String fName = getName();
    assertNotNull(fName);
    Method method = null;
    try {
        // use getMethod to get all public inherited
        // methods. getDeclaredMethods returns all
        // methods of this class but excludes the
        // inherited ones.
        method = getClass().getMethod(fName, (Class[]) null);
    } catch (NoSuchMethodException e) {
```

```

        fail("Method \""+fName+"\" not found");
    }
    //判断方法是否是
    public的

    if (!Modifier.isPublic(method.getModifiers())) {
        fail("Method \""+fName+"\" should be public");
    }
    int runCount = 1;
    boolean isRepetitive = false;
    if (method.isAnnotationPresent(FlakyTest.class)) {
        runCount = method.getAnnotation(FlakyTest.class).tolerance();
    } else if (method.isAnnotationPresent(RepetitiveTest.class)) {
        runCount = method.getAnnotation(RepetitiveTest.class).numIterations();
        isRepetitive = true;
    }
    if (method.isAnnotationPresent(UiThreadTest.class)) {
        final int tolerance = runCount;
        final boolean repetitive = isRepetitive;
        final Method testMethod = method;
        final Throwable[] exceptions = new Throwable[1];
        getInstrumentation().runOnMainSync(new Runnable() {
            public void run() {
                try {
                    runMethod(testMethod, tolerance, repetitive);
                } catch (Throwable throwable) {
                    exceptions[0] = throwable;
                }
            }
        });
        if (exceptions[0] != null) {
            throw exceptions[0];
        }
    } else {
        runMethod(method, runCount, isRepetitive);
    }
}

```

具体的失败重试则是在runMethod（）方法中实现的，通过try catch的形式，如果捕获到测试用例执行过程中的异常，例如用例断言失败，则判断runCount是否小于tolerance的值以及用isRepetitive参数来决定是否进行重跑。

代码清单9-8 InstrumentationTestCase中的runMethod（）源码实现

```

private void runMethod(Method runMethod, int tolerance, boolean isRepetitive)
throws Throwable {
    Throwable exception = null;
    int runCount = 0;
    do {
        try {
            //调用执行该方法

            runMethod.invoke(this, (Object[]) null);
            exception = null;
        } catch (InvocationTargetException e) {
            e.fillInStackTrace();
            exception = e.getTargetException();
        } catch (IllegalAccessException e) {
            e.fillInStackTrace();
            exception = e;
        } finally {
            runCount++;
            // Report current iteration number, if test is repetitive
            if (isRepetitive) {
                Bundle iterations = new Bundle();
                iterations.putInt("currentiterations", runCount);
                getInstrumentation().sendStatus(2, iterations);
            }
        }
    } while ((runCount < tolerance) && (isRepetitive || exception != null));
    if (exception != null) {
        throw exception;
    }
}

```

了解了runTest模式后，就可以自定义地实现更接近业务的失败重试及截图模式。可以编写继承自InstrumentationTestCase的抽象类，覆写runTest（）方法，其中可以对捕获到的异常做分类处理、进行序列化截图等，大致实现思路如代码清单9-9所示。

代码清单9-9 继承自InstrumentationTestCase覆写runTest（）方法

```

@Override
protected void runTest() throws Throwable {
    String testMethodName = getName();
    String currentTestClass = getClass().getName();
    LogUtils.logD(TAG, "currentTestClass:" + currentTestClass);
    boolean isScreenShot = true;
    boolean isScreenShotWhenPass = false;
    boolean isUseFullScreen = true;

```

```

    long startTime = 0;
    long endTime = 0;
    currentActivity = getActivity().getClass().getSimpleName();
    LogUtils.logD(TAG, "currentActivity:" + currentActivity);
    Holo holo = new Holo(getInstrumentation(), getActivity());
    Method method = getClass().getMethod(getName(), (Class[]) null);
    //定义重试次数

    int retrytime = DEFAULT_RETRY_TIME;
    if (method.isAnnotationPresent(RetryTest.class)) {
        //重试次数可以通过注解

@RetryTest(retrytime=1)设置

        retrytime = method.getAnnotation(RetryTest.class).retrytime();
        isScreenShot = method.getAnnotation(RetryTest.class).isScreenShot();
        isUseFullScreen =
method.getAnnotation(RetryTest.class).isUseFullScreen();
    }
    int runCount = 0;
    do {
        try {
            holo.goBackToActivity(currentActivity);
            if(runCount > 0){
                holo.stopScreenshotSequence();
            }
            //开启序列化的截图功能

            holo.startScreenshotSequence(endTime, 5, testMethodName,
currentTestClass);
        }
        startTime = SystemClock.uptimeMillis();
        //执行用例，即执行测试类中的

public修饰的

test开头的方法

        super.runTest();
        endTime = SystemClock.uptimeMillis() - startTime;
        LogUtils.logD(TAG, "run test" + testMethodName + ",testcase pass with
time cost:" + endTime);
        if(isScreenShotWhenPass){
            holo.takeSpoonScreenShot(testMethodName,currentTestClass,testMethodName,DEFAULT_Q
UALITY);
        }
        holo.finishOpenedActivitiesExcept(currentActivity);
        break;
    } catch (Throwable e) {
        //捕获到用例执行过程中有异常（

Throwable包括断言失败时的抛出）时，可以进行各类处理

        LogUtils.logI(TAG, e);holo.takeSpoonScreenShot("current_failed_img",
currentTestClass,testMethodName,DEFAULT_QUALITY);
        //对

```

SecurityException异常进行特殊处理

```
        if(e.getMessage() !=null &&
e.getMessage().contains("SecurityException")){
            //尝试解除屏幕锁定

SelfProcessUtils.wakeup(getInstrumentation().getTargetContext().getApplicationCon
text());
            SuperManager superManager = new SuperManager();
            superManager.goBack();
        }
        //检查

Wi-Fi状态

        checkWifiStat(isUseFullScreen);
        if(retrytime>1 && runCount<retrytime-1){
            runCount++;
            endTime = SystemClock.uptimeMillis() - startTime;
            LogUtils.logD(TAG, "run test" + testMethodName + ",testcase
failed with time cost:" + endTime);
            continue;
        }else {
            if(isScreenShot){
                LogUtils.logD(TAG, "takeScreenshot:");
                holotakeSpoonScreenShot(testMethodName,currentTestClass,testMethodName,DEFAULT_Q
UALITY);
            }
            intentPerformance.putExtra("isStart", false);
            getActivity().sendBroadcast(intentPerformance);
            holofinishOpenedActivitiesExcept(currentActivity);
            //重试过后仍失败的，需要最后抛出

            throw e;
        }
    }
} while (runCount < retrytime);
}
```

在代码清单9-9中，使用了自定义的takeSpoonScreenShot（String tag、String testClassName、String testMethodName、int quality）方法，结合了Robotium中序列化截图方法与Spoon中截图的命名规范，用于在用例失败后，对重试过程进行抽样截图。在用例执行失败后，也可以进行网络状态检查，当网络未连接时可自动进行网络重连等，以减少用例因网络偶然性因素导致的误报现象。使用覆写runTest（）方法

的形式，也可以增加更多自定义的操作，例如在用例执行前开启性能监控、代码覆盖收集等，以最大化地支撑业务需要。

9.3.3 结合Spoon生成汇总报告

如9.3.1节所介绍的，Spoon会生成类似单元测试形式的XML报告文件，因此其他测试平台可以通过解析junit-reports目录下的XML报告获取用例执行的详细数据，对每次的测试进行入库存储，积累日常的测试数据，生成历史记录の测试报告页面，如图9-10所示。

应用宝自动化用例数据（默认显示主干分支）

Show 10 entries

Search:

ID	svn版本	数字版本号	失败用例数	通过用例数	总用例数	创建时间	详情	报告	监控	协议监控
231	r116378	6.3.0.5315	2	122	124	2016-02-05 04:33:25.0				
230	r116378	6.3.0.5315	1	138	139	2016-02-04 04:40:18.0				
229	r116232	6.3.0.5275	1	138	139	2016-02-03 04:43:32.0				
228	r115865	6.3.0.5136	0	139	139	2016-02-02 04:36:27.0				
227	r115865	6.3.0.5136	0	139	139	2016-02-01 04:40:56.0				

图9-10 积累日常的测试数据

9.4 Robotium跨应用

使用Robotium编写的自动化测试用例是基于Instrumentation的，在执行过程中将测试代码注入被测应用程序所在的进程，即测试代码与被测应用运行于同一进程，而出于安全方面的考虑，Android中的普通应用进程不允许发送类似KeyEvent的事件。因此，当测试过程中应用跳转至第三方应用或系统界面，此时仍调用clickOnView（View view）方法时，则会报SecurityException的异常。基于Instrumentation的测试框架带来了诸多优势的同时，却也有天生无法跨应用这一劣势，而在实际项目中还存在许多需要跨应用的测试场景，本节介绍结合UIAutomator及UIAutomation实现跨应用的方法，以便基于Robotium的自动化测试适用于更多测试场景。

dump获取控件再结合adb shell发送模拟操作来实现一个基本的自动化测试框架。

基于此思路的方案在开源界已有简单的实现，项目地址为：

<https://github.com/gb112211/Adb-For-Robotium>。

此方案可以实现跨应用，但缺点也较为明显，由于需要不断地dump界面并解析，在执行速度及稳定性上无法得到保证，且还需要手机拥有ROOT权限。

9.4.2 UIAutomator结合Instrumentation模式

2015年3月，Android Developers团队宣布了UIAutomator 2.0版本的发布，这个版本最重要的特征就是UIAutomator终于可以基于Instrumentation了，使用Instrumentation test runner即可运行UIAutomator。反过来，即在基于Instrumentation的Test中也能使用UIAutomator。因此测试工程可同时使用Robotium和UIAutomator进行更丰富的测试。

新版的UIAutomator随Android Support Repository发布，可通过SDK Manager下载，以2.1.0版本为例，位于如下代码示例所示的路径中：

```
%ANDROID_HOME%\extras\android\m2repository\com\android\support\test\
uiautomator\uiautomator-v18\2.1.0
```

新的测试支持库基本都是基于Android Studio库的，文件以aar结尾而非以jar结尾，本小节为方便在Eclipse中介绍，需要将aar转化成jar。

如图9-12所示，使用压缩工具打开uiautomator-v18-2.1.0.aar文件，里面的classes.jar文件就是可用于Eclipse的UIAutomator jar包。提取出该classes.jar文件并重命名为方便记忆的jar包文件，导入到使用了Robotium的测试工程即可。

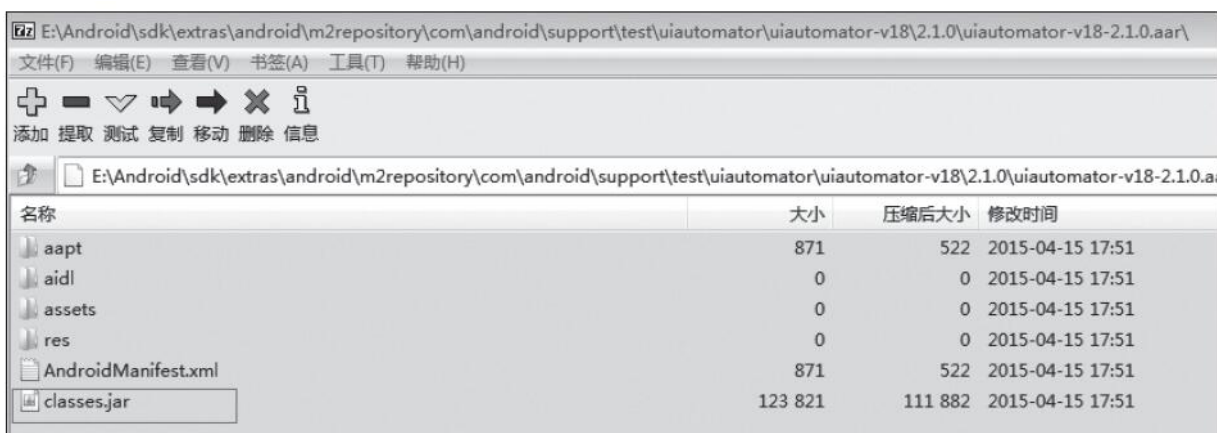


图9-12 解压aar文件

如图9-13所示，应用宝在通知栏中开启了快捷工具栏，测试此功能时需要开启通知栏，并点击工具栏中的按钮，这样的操作仅通过Robotium框架是无法完成的，此时就可以结合UIAutomator来实现。

UIAutomator发布2.0版本后，可以通过传入Instrumentation对象获得UIDevice对象。通过UIDevice对象可以完成点击Home键、打开通知栏，并可以通过UIDevice的findObject方法根据文本、资源ID等查找控件，然后通过UIObject对象完成点击操作。结合UIAutomator的测试示例如代码清单9-10所示。

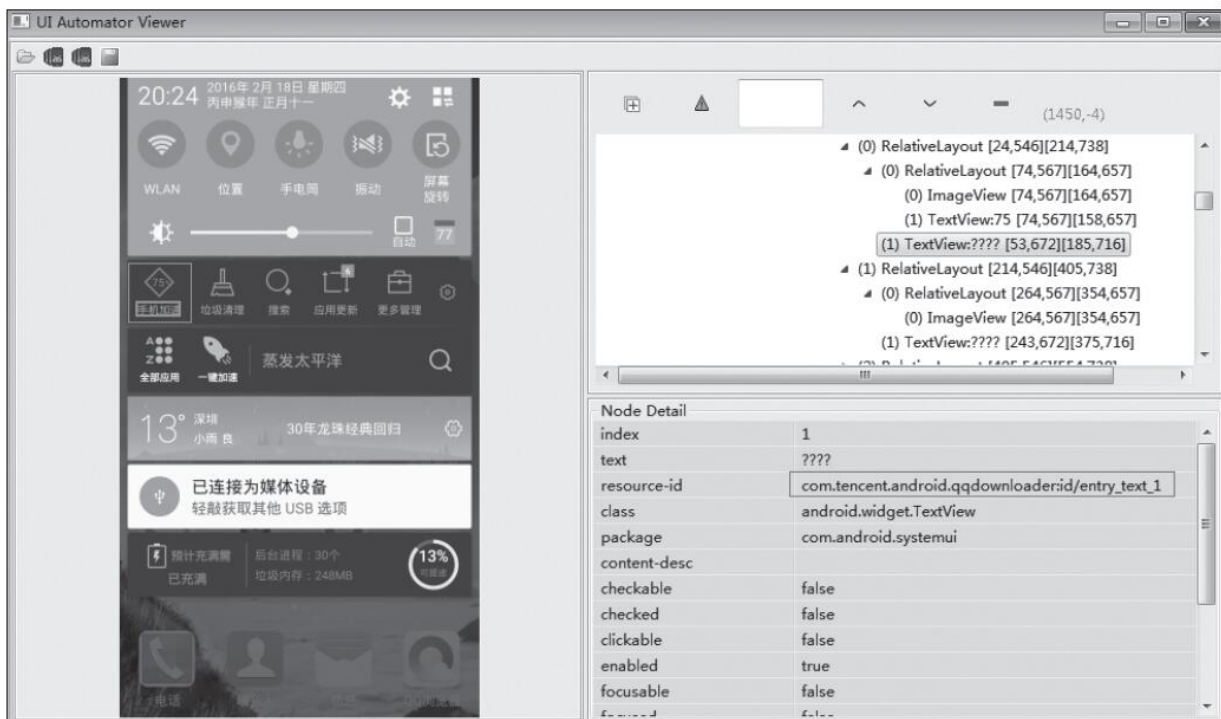


图9-13 应用宝快捷工具栏

代码清单9-10 结合UIAutomator的测试示例

```
package com.tencent.assistant.qa.checklist.nuclear;
import android.support.test.uiautomator.UiDevice;
import android.support.test.uiautomator.UiObjectNotFoundException;
import android.support.test.uiautomator.UiSelector;
import android.test.ActivityInstrumentationTestCase2;
import com.robotium.solo.Solo;
import com.tencent.assistant.qa.constant.YYBConstant;
import com.tencent.assistant.qa.util.LogUtils;
/**
 * 测试快捷工具栏
 */
, 示例

 * @author hangtechen
 */
public class QuickToolBarTestDemon extends ActivityInstrumentationTestCase2{
    private static final String LAUNCHER_ACTIVITY_FULL_CLASSNAME =
YYBConstant.LAUNCHER_ACTIVITY;
    private static Class launcherActivityClass;
    static{
        try
        {
            launcherActivityClass=Class.forName(LAUNCHER_ACTIVITY_FULL_CLASSNAME);
        } catch (ClassNotFoundException e){
```

```

        throw new RuntimeException(e);
    }
}
private static final String TAG=QuickToolBarTestDemon.class.getSimpleName();
private Solo solo;
public QuickToolBarTestDemon() {
    super(launcherActivityClass);
}
@Override
public void setUp(){
    try {
        super.setUp();
    } catch (Exception e) {
        LogUtils.logE(TAG, e);
    }
    solo = new Solo(getInstrumentation(), getActivity());
}
@Override
public void tearDown(){
    hideNotification();
    solo.finishOpenedActivities();
    try {
        solo.finalize();
    } catch (Throwable e1) {
        LogUtils.logE(TAG, e1);
    }
    solo = null;
    try {
        super.tearDown();
    } catch (Exception e) {
        LogUtils.logE(TAG, e);
    }
}
/**
 * 测试快捷工具栏中的手机加速功能

```

</br>

```

    * 1. 开启快捷工具栏

```

</br>

```

    * 2. 进入快捷工具栏，点击手机加速按钮，等待加速完成

```

</br>

```

    * 3. 断言是否出现加载完成的动画

```

</br>

```

    */
    public void test76417919_QuickToolBar_ClickAccelerate(){
        hideNotification();
        openQuickToolBar();    //进入设置页开启快捷工具栏，此处为伪代码

        checkQuickToolBarByText("手机加速

");
        //完成点击操作后，此处增加操作后的期望断言等

    }
    private boolean checkQuickToolBarByText(String item){
        boolean isFind = false;
        //通过

```

Instrumentation获取

UiDevice单例

```
        UiDevice uiDevice = UiDevice.getInstance(getInstrumentation());
        uiDevice.pressHome();
        solo.sleep(1000);
        uiDevice.openNotification();
        solo.sleep(1000);
        try {
            uiDevice.findObject(new UiSelector().text(item)).click();
            isFind = true;
        } catch (UiObjectNotFoundException e) {
            LogUtils.logE(TAG, e);
        }
        return isFind;
    }
    private void hideNotification(){
        UiDevice uiDevice = UiDevice.getInstance(getInstrumentation());
        if(uiDevice.getCurrentPackageName().equals("com.android.systemui")){
            uiDevice.pressBack();
        }
    }
}
```

代码清单9-10中使用的findObject方法得到的是UIObject对象，此外也可以通过By的方式获取UIAutomator中的UIObject2对象，例如：

```
uiDevice.findObject(By.res("com.tencent.android.qqdownloader",
"entry_text_1")).click();
```

UIAutomator2.0还有许多更丰富、更强大的功能，这里就不再一一介绍了。总之，通过与Instrumentation结合可以方便地在测试工程中完成跨应用的操作，进行更丰富的测试。

9.5 代码覆盖率

9.5.1 覆盖率定义

作为一个测试人员，保证产品的软件质量是其工作的首要目标。为了这个目标，测试人员常常会通过很多手段或工具来加以保证，覆盖率就是其中比较重要的一个。

我们通常会将测试覆盖率分为两个部分，即“需求覆盖率”和“代码覆盖率”。

需求覆盖率：指测试人员对需求的了解程度，根据需求的可测试性拆分成各个子需求点，来编写相应的测试用例，最终建立一个需求和用例的映射关系，以用例的测试结果来验证需求的实现，可以理解为黑盒覆盖。

代码覆盖率：为了更加全面地覆盖，我们可能还需要理解被测程序的逻辑，需要考虑到每个函数的输入与输出、逻辑分支代码的执行情况，这个时候我们的测试执行情况就以代码覆盖率为衡量，可以理解为白盒覆盖。

以上两者完全可以相辅相成，用代码覆盖结果反向地检验需求覆盖（用例）的测试是否充分完整。

那么如何做覆盖率测试呢？本节会先简单介绍一下比较主流的几种覆盖率工具，然后选取JaCoCo（适合Java的程序）进行详细介绍，其他工具思路大体是相同的，读者可以举一反三，根据自己项目的特点选取合适的覆盖率工具。

9.5.2 覆盖率工具

1.Javascript测试覆盖率工具

JSCoverage: 一个用于度量Javascript程序的代码覆盖率的工具, JSCoverage支持IE6、IE7、Firefox2、Firefox3、Opera、Safari等流行的浏览器, 支持Windows平台和Linux平台, JSCoverage是开源软件, 官方网站为: <http://siliconforks.com/jscoverage/>。

2.Java测试覆盖率工具

Emma: 离线插桩模式, 即先编译出class文件, 然后插桩, 打包运行。不支持分支覆盖率, 其使用手册地址为:

http://emma.sourceforge.net/reference_single/reference.html。

JaCoCo: 特色是引入agent, 支持在线插桩模式, 即在class加载的时候即时插桩, 同时也支持离线插桩, 具有丰富的dump机制, 支持分支覆盖率, 运行速度比较快。其使用地址为:

<http://eclemma.org/JaCoCo/index.html>。支持gradle方式, 我们在Android覆盖率方面选用的工具为JaCoCo, 优势主要集中在两点: 一是JaCoCo社区比较活跃, 它是原Emma团队新推出的覆盖率工具, Emma项目已经很久没有更新了; 二是JaCoCo比Emma多了分支覆盖。

Coverlipse: 一个Eclipse的Code coverage插件。

Cobertura: 一种开源工具，它通过检测基本的代码，并观察在测试包运行时执行了哪些代码和没有执行哪些代码，来测量测试覆盖率。除了找出未测试到的代码并发现bug外，Cobertura还可以通过标记无用的、执行不到的代码来优化代码，还可以提供API实际操作的内部信息。

3..NET测试覆盖率工具

Clover.NET: Visual Studio的代码覆盖率统计工具，其官方网站为: <http://www.cenqua.com/clover.net/>。

NCover官方网站为: <http://ncover.org/>。

PartCover: 与NCover非常相似，PartCover是针对.NET的一个开源代码覆盖工具。它包括了一个控制台应用程序、GUI覆盖浏览器，以及用在CC.NET中的xsl转换。

4.C/C++测试覆盖率工具

Bullseye Coverage: Bullseye公司提供的一款C/C++代码覆盖率测试工具，除了支持各种UNIX下的编译器之外，在Windows下还支持VC、Borland C++、Gnu C++、Inter C++。提供的代码覆盖率是分支覆

盖率而不是一般的代码覆盖率，个人认为分支覆盖率比代码覆盖率更好。Bullseye Coverage可以从<http://www.bullseye.com/> 上获取。

5.Ruby测试覆盖率工具

rcov: 一个用于诊断Ruby代码覆盖率的工具，它最主要的用途就是确定单元测试是否覆盖到所有代码，rcov使用一个经过优化的C运行，因此性能相当惊人，同时它还提供多种格式的输出。

6.其他

AutomatedQA公司的AQtime。AQtime运行在Windows平台上，它支持.NET应用和非.NET应用，但不支持Java应用。AQtime除了包含代码覆盖率监测以外，还包括性能监视等功能。

DevPartner Studio的Web script Coverage工具。该工具主要是收集Web客户端script脚本覆盖率的。

9.5.3 JaCoCo介绍与实践

1.JaCoCo简介

JaCoCo是一个开源的覆盖率工具（官网地址：<http://www.eclemma.org/JaCoCo/>），它针对的开发语言是Java，其使用方法很灵活，可以嵌入Ant、Maven中，可以作为Eclipse插件，还可以使用其JavaAgent技术监控Java程序等。很多第三方的工具提供了对JaCoCo的集成，如sonar、Jenkins等。

JaCoCo包含了多种尺度的覆盖率计数器，包含指令覆盖（Instructions, C0coverage）、分支覆盖（Branches, C1coverage）、圈复杂度（CyclomaticComplexity）、行覆盖（Lines）、方法覆盖（non-abstract methods）、类覆盖（classes）等（详细内容后面会有介绍）。

我们可以先看看其覆盖率的结果展现，如图9-14所示。

```

474.     public void freshView(int type) {
475.         int size = 0;
476.         ◆ if(this.isSameTagApps) {
477.             size = mSameTagApps.size();
478.         } else {
479.             size = apps.size();
480.         }
481.         ◆ for(int i=0; i< size; i++) {
482.             ◆ if (appLayouts[i] != null) {
483.                 appLayouts[i].setVisibility(View.VISIBLE);
484.             }
485.             RecommendAppInfo detail = apps.get(i);
486.             appImages[i].updateImageView(detail.iconUrl, R.drawable.pic_defaule, TXImageViewType.NETWORK_IMAGE_ICON);
487.             try {
488.                 appTexts[i].setText(Html.fromHtml(detail.appName));
489.             } catch (NullPointerException e) {
490.                 e.printStackTrace();
491.             }
492.             ◆ if(!TextUtils.isEmpty(getAppDescription(i))) {
493.                 appReasons[i].setText(Html.fromHtml(getAppDescription(i)));
494.                 appReasons[i].setVisibility(View.VISIBLE);
495.             } else {
496.
497.                 appReasons[i].setVisibility(View.GONE);

```

图9-14 覆盖率报告结果部分截图

标绿色的为行覆盖充分（如481~488行），标红色的为未覆盖的行（477行、489行、490行），标黄色菱形的为分支部分覆盖（476行、482行），标绿色菱形的为分支完全覆盖（481行）。通过这个报告结果就可以知道代码真实的执行情况，便于我们分析评估结果。



注意 截图是带有颜色的，如果图打印成黑白色，请读者参考括号里的行号，这里主要是描述一个点，JaCoCo对覆盖率结果会有不同样式的展现。

2.JaCoCo知识点

JaCoCo使用一系列的不同的计数器来做覆盖率的度量计算，所有这些计数器都是从Java的class文件中获取信息，这些class文件里面可以包含调试的信息。即使在没有源码的情况下，这种方法也可以实时有

效地对应用程序进行度量和分析，但看不到具体源码的覆盖率执行情况。如果做详细的覆盖率分析，必须指定源码，这样可视化到每一行代码的粒度（图9-14）。前提是这些class文件必须使用调试信息来编译，这样才可以计算行的覆盖率和提供出源码的高亮。

JaCoCo主要有以下几个纬度数据：

1) 指令覆盖

JaCoCo最小的计数单元是单个Java二进制代码指令。指令覆盖率提供了代码是否被执行的信息。这个度量完全独立于源码格式，即使class文件里面没有调试信息也总是可用的。

2) 分支覆盖

JaCoCo也计算分支的覆盖率，包括所有的if和switch语句。这个度量计算一个方法里面的总分支数，确定执行和不执行的分支数量。分支覆盖率总是可用的，即使class文件里面没有调试信息。注意，异常处理是不在分支度量里面统计的。如果class文件使用调试信息编译的话，产生的覆盖率可以映射到源码行并且高亮提示：

- 没有覆盖：在这一行中没有分支被执行（红色方块）；

- 部分覆盖：这一行的分支中只有一部分被执行（黄色方块）；

·完全覆盖：这一行的所有分支都被执行（绿色方块）。

3) 圈复杂度

JaCoCo同样可以为每一个非抽象方法计算复杂度，最终计算出类、包和组的复杂度。由McCabe1996可知圈复杂度的定义是，在（线性）组合中，计算在一个方法里面所有可能路径的最小数目。所以复杂度可以作为度量单元测试是否完全覆盖所有场景的一个依据，复杂度即使是在没有调试信息的情况下也可以计算。

圈复杂度 $V(G)$ 的正式定义是基于方法的控制流图的有向图表示的：

$$V(G) = E - N + 2$$

E 是边界的数量， N 是节点的数量。JaCoCo基于下面的方程来计算复杂度， B 是分支的数量， D 是决策点的数量：

$$V(G) = B - D + 1$$

基于每个分支的被覆盖情况，JaCoCo也为每个方法计算覆盖和缺失的复杂度。缺失的复杂度同样表示测试案例没有完全覆盖到这个模块。注意JaCoCo不将异常处理作为分支，try/catch块也同样不增加复杂度。

4) 行覆盖

所有的class文件使用debug信息编译之后，就可以计算行的覆盖率信息。一行源代码是否被执行，要看这一行中是否至少有一个指令被执行。

由于实际上一行代码一般被编译成多个二进制代码指令，这样源码在高亮显示时，会显示成三种不同的状态：

- 没有覆盖：这一行中没有指令被执行（红色背景）；
- 部分覆盖：这一行中只有一部分指令被执行（黄色背景）；
- 完全覆盖：这一行中所有指令都被执行（绿色背景）。

5) 方法覆盖

每一个非抽象方法至少包含一个指令，一个方法是否执行取决于方法中是否有至少一个指令被执行。在JaCoCo中，构造器和静态初始化同样会像方法一样统计，其中一些方法可能没有可以直接对应的源码，比如默认构造器或常量的初始化命令。

6) 类覆盖

一个方法是否执行取决于类中是否至少有一个方法被执行，注意JaCoCo认为构造器和静态初始化都是方法，Java的接口一般包含静态

初始化，所以接口也同样被认为是可执行的类。

7) 包覆盖

包覆盖描述一个package的覆盖程度，在XML报告中有数据体现，HTML报告中只能去综合评估。

3.JaCoCo实践

本小节JaCoCo实践包括代码插桩与编译打包、执行测试并收集覆盖率结果、生成覆盖率报告、分析覆盖率结果等主要步骤。

步骤1：代码插桩与编译打包。

以Android项目使用Ant进行编译为例，可以修改build，将JaCoCo的部分添加进去，这样打成的包即为覆盖率包。

(1) 文件开头的命名空间加入以下代码。

```
xmlns:JaCoCo="antlib:org.JaCoCo.ant"
```

(2) 引入JaCoCo的jar和相关定义。

```
<taskdef uri="antlib:org.JaCoCo.ant"
  resource="org/JaCoCo/ant/antlib.xml">
<classpath path="${basedir}/libs/JaCoCoant.jar" />
```

(3) 重新定义class文件生成路径。

```
<property name="classes_instr" value="${temp}/classes_instr" />
```

(4) 修改编译节点，进入class文件，注入以下代码。

```
<JaCoCo:instrument destdir="${classes_instr}">
  <fileset dir="${classes}" includes="**/*.class" />
</JaCoCo:instrument>
```

(5) 修改打包，主要是指指定JaCoCo编译后的类路径。

```
<jar basedir="${classes_instr}" destfile="temp.jar" />
```

(6) 修改混淆，增加混淆所需要的代码如下：

```
<arg value="-libraryjars ${lib}/JaCoCoant.jar" />
<arg value="-libraryjars ${lib}/JaCoCoagent.jar" />
```

(7) 其他。如果项目有clean、remove等操作，需要根据上面的修改做相应的清理工作。

(8) 执行Ant，编译生成新的带有覆盖率的测试包。

若是使用Gradle进行编译，则过程包括以下几方面。

·在项目的build.gradle引入插件。

```
apply plugin:
    "jacoco"
```

·注明使用的版本号，如下：

```
jacoco {  
    version "0.7.4.201502262128"  
}
```

·声明一个gradle task。

```
task jacocoTestReport(type:JacocoReport,dependsOn:"connectedAndroidTest"){  
    group = "Reporting"  
    description = "Generate Jacoco coverage reports after running tests."  
    reports{  
        xml.enabled = false  
        html.enabled = true  
        csv.enabled = false  
    }  
    classDirectories = fileTree(  
        dir : "$buildDir/intermediates/classes/debug",  
        excludes : [ '**/*Test.class', '**/R.class', '**/R$*.class', '**/BuildConfig.*',  
                    '**/Manifest*.*' ] )  
    def coverageSourceDirs = ['src/main/java']  
    additionalSourceDirs = files(coverageSourceDirs)  
    sourceDirectories = files(coverageSourceDirs)  
    additionalClassDirs = files(coverageSourceDirs)  
    executionData = files("$buildDir/outputs/code-coverage/connected/coverage.ec")  
}
```

·打开testCoverageEnabled。需要注意的是，打开该属性的话，在断点调试的时候会导致方法参数值丢失（看不到），所以在调试的时候要记得把它关掉。

```
buildTypes {  
    debug{  
        testCoverageEnabled true  
    }  
}
```

完整的Gradle配置如代码清单9-11所示。

代码清单9-11 完整的Gradle配置

```
apply plugin: 'com.android.library'
apply plugin: 'jacoco'
android {
    compileSdkVersion 22
    buildToolsVersion '22.0.1'
    defaultConfig {
        minSdkVersion 8
        targetSdkVersion 22
        versionCode 1
        versionName "1.0"
    }
    buildTypes
    {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
        debug{
            testCoverageEnabled true
        }
    }
    lintOptions {
        abortOnError false
    }
    packagingOptions {
        exclude 'META-INF/NOTICE'
        exclude 'META-INF/LICENSE'
    }
    jacoco{
        version "0.7.4.201502262128"
    }
}
task jacocoTestReport(
    type:JacocoReport,dependsOn:"connectedAndroidTest"){
    group = "Reporting"
    description = "Generate Jacoco coverage reports after running tests."
    reports{
        xml.enabled = false
        html.enabled = true
        csv.enabled = false
    }
    classDirectories = fileTree(
        dir : "$buildDir/intermediates/classes/debug",
        excludes : [ '**/*Test.class', '**/R.class', '**/R$.class', '**/BuildConfig.*',
            '**/Manifest.*' ]
    )
    def coverageSourceDirs = ['src/main/java']
    additionalSourceDirs = files(coverageSourceDirs)
    sourceDirectories = files(coverageSourceDirs)
    additionalClassDirs = files(coverageSourceDirs)
    executionData = files("$buildDir/outputs/code-coverage/connected/coverage.ec")
}
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.2.1'
}
```

步骤2: 执行测试并收集覆盖率结果。

收集覆盖率的方式主要是反射调用JaCoCo API的dump方法，以文件追加的方式在手机上写覆盖率结果，实现方法如代码清单9-12所示。

代码清单9-12 反射调用JaCoCo API的dump方法

```
private void dumpCoverageJacoco(boolean reset){
    String absFilePath = sdCard + (sdCard.endsWith(File.separator) ? "coverage":
    File.separator + "coverage") + File.separator + getName()+".ec";
    File coverageFile = new File(absFilePath);
    try {
        Class classAgentOptions =
        Class.forName("org.jacoco.agent.rt.internal_b0d6a23.core.runtime.AgentOptions");
        //Get setDestfile method in AgentOptions class
        Method methodSetDestFile =
        classAgentOptions.getMethod("setDestfile",String.class);
        //Get FileOutputStream class
        Class classFileOutput =
        Class.forName("org.jacoco.agent.rt.internal_b0d6a23.output.FileOutput");
        //Get field "File destFile" in FileOutput class
        Field fieldFile = classFileOutput.getDeclaredField("destFile");
        fieldFile.setAccessible(true);
        //Get Agent singleton by getAgent method in RT class
        Class<?> RT = Class.forName("org.jacoco.agent.rt.RT");
        Method methodGetAgent = RT.getMethod("getAgent");
        Object objAgent = methodGetAgent.invoke(null);
        //Get Agent Class
        Class classAgent =
        Class.forName("org.jacoco.agent.rt.internal_b0d6a23.Agent");
        //Get field "AgentOptions options" and "FileOutput output" in Agent Class
        Field fieldOptions = classAgent.getDeclaredField("options");
        Field fieldOutput = classAgent.getDeclaredField("output");
        fieldOptions.setAccessible(true);
        fieldOutput.setAccessible(true);
        //Get options/output object referenced by Agent singleton
        Object objOptions = fieldOptions.get(objAgent);
        Object objOutput = fieldOutput.get(objAgent);
        //change destFile attribute in options object by setDestfile method
        methodSetDestFile.invoke(objOptions,absFilePath);
        //change field "File destFile" in output object
        File destFile = new File(absFilePath).getAbsoluteFile();
        fieldFile.set(objOutput,destFile);
        //dump
        Method methodDump = classAgent.getMethod("dump",boolean.class);
        methodDump.invoke(objAgent,reset);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

步骤3: 生成覆盖率报告。

通过编写report的build脚本来生成报告结果，build脚本如代码清单9-13所示，配置好后，执行Ant即可。

代码清单9-13 生成覆盖率报告的Ant脚本

```
xmlns:JaCoCo="antlib:org.JaCoCo.ant"
<project xmlns:jacoco="antlib:org.jacoco.ant" name="Example Ant Build with JaCoCo
Offline Instrumentation" default="rebuild">
  <property name="result.dir" location="."/>
  <property name="src.dir" location="./src"/>
  <property name="result.classes.dir" location="${result.dir}/classes"/>
  <property name="result.report.dir" location="${result.dir}/report"/>
  <property name="result.report.xml" location="${result.dir}/result.xml"/>
  <property name="result.exec.file"
location="${result.dir}/testApkMgr_ClickCleanBtn_ShouldCleanApks.ec"/>
  <!-- Step 1: Import JaCoCo Ant tasks -->
  <taskdef uri="antlib:org.jacoco.ant" resource="org/jacoco/ant/antlib.xml">
    <classpath path="./libs/jacocoant.jar"/>
  </taskdef>
  <target name="report">
    <jacoco:report>
      <executiondata>
        <file file="${result.exec.file}"/>
      </executiondata>
      <structure name="JaCoCo Ant Example">
        <classfiles>
          <fileset dir="${result.classes.dir}"/>
        </classfiles>
        <sourcefiles encoding="UTF-8">
          <fileset dir="${src.dir}" includes="**/*.java" />
        </sourcefiles>
      </structure>
      <html destdir="${result.report.dir}"/>
      <csv destfile="${result.report.dir}/report.csv"/>
      <xml
destfile="${result.report.xml}/testApkMgr_ClickCleanBtn_ShouldCleanApks.xml"/>
    </jacoco:report>
  </target>
  <target name="rebuild" depends="report"/>
</project>
```

若是以Gradle方式，则打开Terminal，并输入命令“gradlew jacocoTestReport”（task名字）执行。

步骤4: 分析覆盖率结果。

网上关于JaCoCo覆盖率报告的分析有不少的文章可以学习，这里主要阐明几个观点。

根据项目的不同，在分析结果前应该先明确以下几点：

(1) 确定改动点的范围，根据这个范围才能有针对性地做分析。

我们不可能对全部代码的覆盖率结果都分析一遍，有针对性地进行分析才能分析到点上，笔者总结了两点分析的过程，分享给大家：

第一，和开发确认好本次修改点或新增点的代码范围，如SVN从版本11765到11899。

第二，针对上面的代码范围来确定我们覆盖率结果的分析范围，这里主要考虑两点，一是代码本身的修改范围，二是修改的代码和其他未修改代码的耦合关系范围。将这部分代码覆盖率结果拿出来作为我们分析的源头。

(2) 改动点是否影响功能逻辑，如果不影响可以忽略。

不是所有的改动点都一定要覆盖到，在分析的过程中要抓住重点，建议梳理出功能的优先级，由高到低去分析，原则上有几个点可以忽略：保护代码（比如非空判断等）、异常和catch部分。

(3) 改动点和其他功能是否存在耦合，如果存在，耦合的部分也要做分析。

点覆盖全了，也要考虑面的覆盖，这样才能真正完全地覆盖。

我们主要从上面几点来分析覆盖率，查漏补缺，这些改动点大部分已经覆盖到了，基本认为应用的主要功能覆盖完全，当然也不是绝对的。在测试过程中，结合FreeTest、探索性测试等手段也是一种不错的选择，切记不要盲目地为了覆盖率而覆盖，覆盖率高并不代表真的覆盖完全了。很多人觉得分析过程是比较痛苦的，不妨把这个过程当作一种锻炼，前面的一切都只是一个铺垫，最关键的就在于分析阶段，一个出色的分析结果可以达到事半功倍的效果。

除了Apache Ant方式外，也可以采用其他方式，例如：

命令行方式：可以参见

<http://www.eclemma.org/JaCoCo/trunk/doc/agent.html>；

Apache Maven方式：可以参见

<http://www.eclemma.org/JaCoCo/trunk/doc/maven.html>；

Eclipse EcJmma Plugin方式：可以参见<http://www.eclemma.org/>。

4.JaCoCo持续集成

这里使用Jenkins进行持续集成，步骤如下：

(1) 在Jenkins上安装JaCoCo的插件，安装完成之后在job的配置项中就会增加JaCoCo相关选项，如图9-15所示。

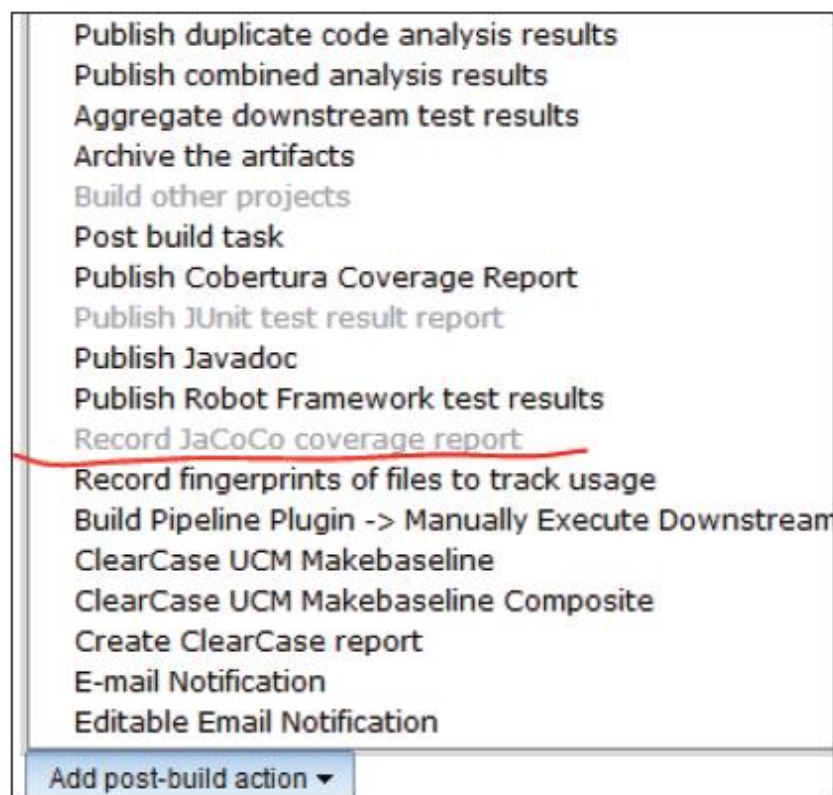


图9-15 Jenkins中的JaCoCo插件

(2) 选择Record JaCoCo coverage report后，在第一个输入框中输入覆盖率文件（exec），第二个输入框中输入class文件目录，第三个输入框中输入源代码文件目录，如图9-16所示。

Record JaCoCo coverage report		
Path to exec files (e.g.: <code>**/target/**/*.exec</code> , <code>**/jacoco.exec</code>)	Path to class directories (e.g.: <code>**/target/classDir</code> , <code>**/classes</code>)	Path to source directories (e.g.: <code>**/mySourceFiles</code>)
<code>**/*.exec</code>	<code>**/build/classes</code>	<code>**/src/java</code>
删除		

图9-16 Jenkins中Record JaCoCo coverage report

(3) 配置好之后进行构建，构建完成之后job首页就会出现覆盖率的趋势图（图9-17），鼠标点击趋势图则可以看到覆盖率详情，包括具体覆盖率数据和源码的覆盖率情况等。



图9-17 覆盖率趋势图

9.5.4 BVT测试与覆盖率结合

1.BVT测试结合JaCoCo覆盖率

结合覆盖率与BVT自动化测试，可以得到每个BVT的用例的覆盖率数据，从而得出几个纬度的结果：

- 可以通过覆盖率报告看出自动化测试用例的覆盖情况，从而调整、优化、补充测试用例。
- 可以动态映射自动化测试用例与源代码的关系，从而可以按方法的调用频繁度来优化代码，优化调用频繁度高的代码，找出冗余代码，等等。需要注意的是，用例和代码的动态映射关系，可能会存在映射到的函数比较多的情况，因此建议根据功能有针对性地筛选出重点函数来做映射。

BVT与覆盖率进行结合，主要包括以下几个步骤。

1) 在BVT用例中插入覆盖率方法

应用宝的BVT用例基于Robotium框架编写，有setUp及tearDown固有生命周期，因此要收集单个用例的覆盖率数据，可以在setUp时重置清理之前旧的覆盖率数据，如代码清单9-14所示。在用例执行过程中收

集到覆盖率数据后，在tearDown中dump出覆盖率数据，如代码清单9-15所示。

代码清单9-14 setUp时清理之前旧的覆盖率数据

```
@Override
protected void setUp() throws Exception {
    super.setUp();
    //反射调用

    JaCoCo API的

    reset方法

    ...
    Method methodDump = classAgent.getMethod("reset");
    methodDump.invoke(objAgent, null)
}
```

代码清单9-15 tearDown时dump出覆盖率数据

```
@Override
protected void tearDown() throws Exception {
    //反射调用

    JaCoCo api的

    dump方法

    ...
    Method methodDump = classAgent.getMethod("dump", boolean.class);
    methodDump.invoke(objAgent, reset);
}
```

2) 执行BVT用例，得到覆盖率

运行BVT的用例，用例执行完成后输出覆盖率文件，一个用例对应一个覆盖率文件（图9-18）。









 test76410817_FloatWindo... (93 KB)	 test76410829_FloatWindo... (94 KB)	 test76410831_FloatWindo... (94 KB)	 test76410859_AppDetail_... (87 KB)
 test76410867_AppDetail_... (87 KB)	 test76410893_MiniDeskto... (87 KB)	 test76410895_MiniDeskto... (88 KB)	 test76410897_MiniDeskto... (88 KB)
 test76410903_AppDetail_... (87 KB)	 test76410913_Search_Ent... (87 KB)	 test76410915_Search_Ent... (87 KB)	 test76410917_Search_Clic... (87 KB)
 test76410919_Search_Clic... (87 KB)	 test76410957_Mgr_Mobil... (92 KB)	 testMgr_ClickAppUninstall... (92 KB)	 testMgr_InstalledAppMan... (92 KB)
 testMgr_InstalledAppMan... (93 KB)	 testSearch_ClearSearchHis... (87 KB)	 testSearch_ClickAppInRes... (87 KB)	 testSearch_ClickTabsInRes... (87 KB)
 testSettingChild_Personal... (83 KB)	 testSettingChild_PhoneMa... (83 KB)	 testSettingChild_Recomm... (83 KB)	 testSettingChild_Software... (83 KB)
 testSetting_AutoDelPacka... (83 KB)	 testSetting_AutoInstall.ec (83 KB)	 testSetting_FloatWindow.ec (83 KB)	 testSetting_NoPictureMod... (83 KB)
 testSetting_WiFiBookingW... (83 KB)			

图9-18 BVT生成的覆盖率文件列表

3) 批量生成覆盖率报告，解析入库

将上面的EC文件批量生成覆盖率报告，生成XML格式的报
告，根据XML的文件格式（图9-19），我们设计出四张表，每个用例的覆盖率文件分别入四张表：testcase表、package表、class表、method表，记录存储的就是XML里面的内容，通过解析程序将这些记录全部入库。

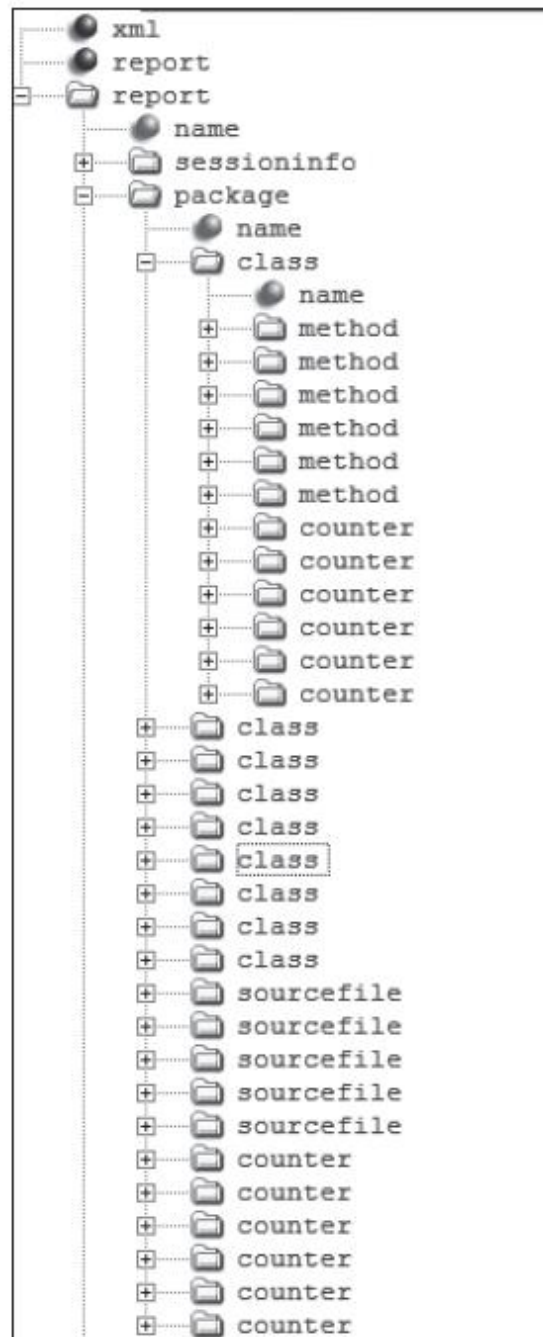


图9-19 JaCoCo生成的XML格式

4) 分析覆盖率结果，得出用例和代码映射关系

我们已经得出每一个BVT用例的覆盖率数据，下面对每一个覆盖率数据结果进行分析，这个是比较关键的点，步骤如下：

- 用例→包→类→方法的覆盖数据，重点找出method表的coverd的数据。

method表数据：筛选出method_coverd=1的所有数据。

- 根据上面的数据再次筛选。

我们发现method_coverd=1的所有数据也非常多，一个用例往往对应几千个方法，这些方法真的都是我们要找的对应的方法吗？答案是否定的。在一个用例的执行过程中，内存中往往会有其他方法在执行，这部分数据也会一起记录进来，我们需要把这些方法剔除出去，比如构造方法、定时任务、Server的触发等，再根据用例对应功能的特点，筛选出属于该功能对应的代码package范围，最终形成一个比较精简的用例和代码映射关系。

2.差异覆盖率和全量覆盖率

根据覆盖率结果，覆盖率可以分为差异覆盖率和全量覆盖率。

- 差异覆盖率：改动点的代码执行覆盖率情况。

差异覆盖率主要是根据开发代码变更的diff差异，得出改动代码的范围，然后根据这个范围有针对性地只生成这部分改动的代码覆盖率结果。通过覆盖率结果反向衡量测试的充分性，更好地和精准评估的测试范围去做比较。

- 全量覆盖率：本次测试代码执行全部覆盖率情况。

全量覆盖率即全部代码的覆盖结果，不一定要全部去分析，只需关注改动部分及其耦合功能的覆盖情况即可。

当BVT用例执行完成并收集到覆盖率后，使用哪种覆盖率是由测试阶段的内容决定的，比如上线前测试、集成或合流阶段，主要关注的是改动点的变化，使用差异覆盖率效果比较理想。如果是新增功能，则使用全量覆盖率比较理想。

3.衡量覆盖率结果

代码覆盖是一种状态指示器，而不是衡量性能或正确性的单元。代码覆盖率是给程序员参考用的，是让程序员发现代码中问题的一种手段，可以发现过时的、未测试的类，还可以发现未经测试执行可能导致问题的路径。在实际项目中，代码覆盖率总是低于100%。取得完全覆盖是不可能的，如果取得，那也是非常罕见的。分析前一定要确定哪些为必须覆盖，哪些为可以或不覆盖，不要为了覆盖而覆盖，代

码逻辑的熟练程度对分析覆盖率会有很大的帮助，一定要先梳理清楚。

4.基于覆盖率数据分析用户喜好

除了本小节介绍的覆盖率常规应用外，还可以进行发散性应用，例如基于覆盖率数据来分析用户喜好。

在应用中我们添加一个测试服务，定时地收集覆盖率数据并上传到后台服务器中。后台收集这部分数据，只取coverage=1（即已覆盖的数据），按其三个纬度存储（包、类、方法），把所有的数据进行一下统计，会得出三个纬度的总数据，格式如下（表格里的数据只是举例，无参考价值）：

总次数	包已覆盖数据	类已覆盖数据	方法已覆盖数据
108	54	32	14

按次数优先级做一个正序，就可以得到该用户操作频繁度由高到低的一个列表，将这些按照我们积累的知识库划分归类，就可以得到该用户功能使用高低的列表。功能覆盖率低的可以考虑淡化和去掉，功能覆盖率高的可以考虑优化和新增功能。

9.5.5 指导建议

代码覆盖率是软件测试中的一种度量手段，主要用来描述程序中源代码被测试的比例和程度。在单元和系统测试过程中，其常常作为衡量测试好坏的指标，甚至在很多情况下用代码覆盖率来考核测试任务完成情况，经常会被要求代码覆盖率必须达到××%以上，才算测试充分。于是，测试人员或者开发人员费尽心思设计案例来覆盖代码，这种用代码覆盖率来衡量的方法，有利也有弊。

以下是给读者的一些建议：

- 覆盖率数据只能代表你测试过哪些代码，不能代表你测好了这些代码。

- 不要过于相信覆盖率数据。

- 不要只拿语句/行覆盖来衡量。

- 路径覆盖率>判断覆盖>语句覆盖。

- 不要盲目地为了提供覆盖率而补充用例，应该想办法设计更好的用例，哪怕多设计的用例对覆盖率提升没有效果。

9.6 本章小结

本章从测试工程、测试用例、测试报告、跨应用、代码覆盖率等多种角度来介绍基于Instrumentation的自动化测试及在应用宝BVT中的应用。在这一过程中涉及Robotium、Spoon、UiAutomator、JaCoCo等技术或框架，从中也可以看出自动化测试在一个项目实际实践中往往是结合多种技术与框架的，而基于Instrumentation的自动化测试，由于测试代码本身以APK形式安装在手机中，测试工程也可以方便地调用Android平台中丰富的类库，例如通过Intent发送广播、创建后台Services进行监控、数据库读写、切换网络等。另外，代码覆盖率不仅可以应用到自动化测试过程中，手工测试也非常适合使用，尤其对于新功能的覆盖，其作用不言而喻。我们只有在实践过程中放开思维，才能创造更多可能。

第10章 兼容性测试实践

本章从三个纬度对兼容性测试进行了介绍。首先介绍兼容性测试定义，然后依次介绍手动测试方法、自动化测试方法、云平台测试方法，如图10-1所示。

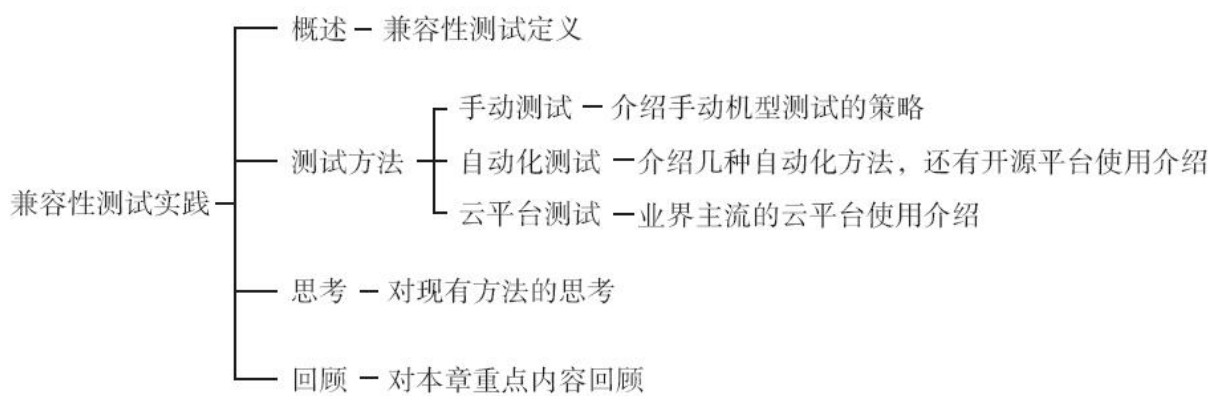


图10-1 本章知识结构图

兼容性测试的方法很多，无论是手动、自动化还是云平台，都只是手段。我们必须从实际收益出发，来选择适合项目本身的方法。

10.1 兼容性测试概述

兼容性测试主要是指测试Android应用的功能，在市面上所有的Android设备上能否正常运行。

为什么主要是Android而不是iOS呢？大家想一下，大部分测试人员所知的iOS设备会超过50款吗？很显然不会超过。但是，你所知的Android设备会少于50款吗？答案很明显了。2015年10月，中国Android手机市场在售机型数量达到1144款，碎片化非常严重，如图10-2所示。

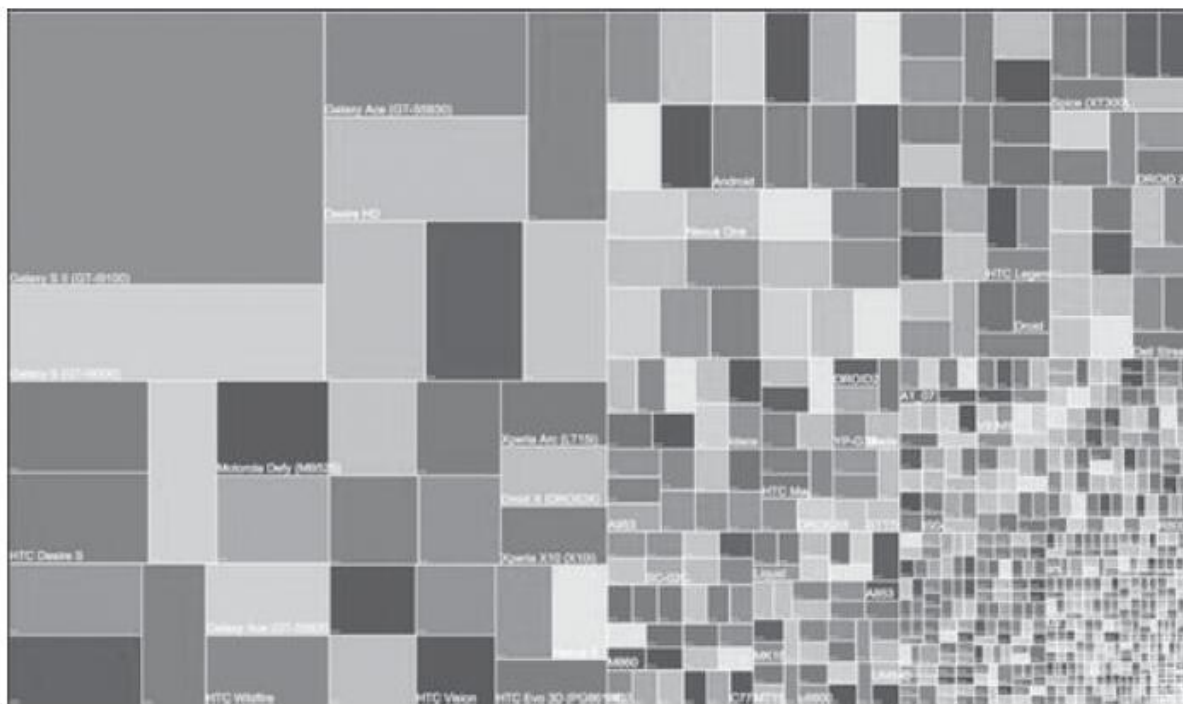


图10-2 2015年10月中国Android机型碎片化程度

之所以做机型兼容性测试，主要原因有以下几方面：

·设备碎片化：2015年Android机型增加了60%，达到18679，这个数字更是2012年的4倍多。

·品牌碎片化：三星占比最高，有43%的份额，中国品牌排名靠前的有华为、联想、中兴、小米、OPPO等。

·系统碎片化：Android的不同版本分布情况严重。

·传感器碎片化：传感器品种越来越丰富。

·屏幕碎片化：Android的屏幕尺寸规格众多。在这种碎片化中，你的App说不好会落到哪个坑里面。也许是某个特殊屏幕分辨率，或者是某个特殊的传感器API。

·动态skia：封装中间层，动态调用系统渲染API，要做机型覆盖。

·静态skia：打包所有系统接口，静态系统渲染API，要做机型覆盖。

·游戏引擎：Canvas游戏、Egret引擎用到GPU的OpenGL接口做硬件加速，要做机型覆盖。

·AndroidL：新系统支持，要做机型覆盖。

特性代码中用到了机器本身的硬件接口，由于机器的多样性、差异性，一旦代码对某类接口有所遗漏或者处理不当，就会出现各种异

常。

10.2 兼容性测试方法

兼容性测试主要有手动测试、自动化测试和云平台测试三种方法。本节我们也分别从这三个方面进行介绍。

10.2.1 手动测试

兼容性测试最简单的，就是在日常手工测试中，按照一定的策略进行测试。具体有哪些策略呢？

（1）TOP机型覆盖。例如，在手机QQ浏览器（Android）测试中，通常笔者在“迭代测试”阶段采用当前Android TOP 50（数据来源：产品经理）机型。在“上线前”测试中，笔者缩小范围，采用TOP20机型进行测试。

（2）差异机型。先分析得出机器差异性在于GPU，再根据对GPU品牌型号的分析，做精准覆盖。例如：

- 高通GPU的机器可以主要覆盖Adreno 200和Adreno 203，基本占高通总数的60%。

- Imagination: GPU的机器主要覆盖SGX544+和SGX531，约占该品牌总数的65%。

- Mali: 覆盖Mali-400MP，占72%。

用上述GPU的机器，在测试中重点覆盖。

(3) 已有BUG分析的机型覆盖。通过对手机QQ浏览器（Android）现有BUG库中机型问题进行归纳汇总，笔者得到了表10-1中的内容。

表10-1 手机QQ浏览器（Android）机型BUG总结

类别	浏览器功能划分	机型覆盖重点
功能类	Canvas 游戏	GPU
	Egret 引擎 Cocos 引擎 Laya 引擎	GPU
	BLINK 内核	GPU
	静态 skia	系统 + 特殊机型库
	动态 skia	系统 + 特殊机型库
	三方字体显示	系统 +TOP 字体
	外文显示	系统
	AndroidL	系统
(续)		
类别	浏览器功能划分	机型覆盖重点
功能类	快速纹理上传	系统
	Flash	2.X 软绘 +4.X 软绘 +4.X 硬绘
	闪屏	分辨率
	自绘框	分辨率
性能类	速度	中低端机器
	内存	中低端机器
	流畅性	中低端机器

10.2.2 自动化测试

现在业界主流机型兼容自动化思路，是利用多机型云平台海量的设备进行被测App的安装卸载、稳定性、功能测试等测试。本节主要介绍自动化实现部分，云平台使用部分在下一节介绍。

1.安装卸载

通过在Android设备上安装被测应用→启动被测应用→卸载被测应用，来检验以下两方面内容。

1) 安装包的安装兼容性

典型例子是，在Android2.X系统上，如果App中method方法数超过65K（65535），就会出现安装失败。根本原因是安装APK的时候，Android2.X的做dexopt使用LinearAlloc固定5MB空间存储方法数，所以有方法数65K（65535）这个限制。具体实现原理方法如下：

通过adb（Android Debug Bridge）进行安装和卸载。例如：安装包test.apk，包名com.sample.app，启动Activity是MainActivity。

安装：adb install test.apk。

启动：adb shell am start-n com.sample.app/.MainActivity。

卸载: adb uninstall test.apk。

覆盖安装: adb install-r test.apk。

通过上述命令, 进行App安装、启动、卸载。观察console输出, 如果是success就是成功, 反之就是失败。同时抓取Logcat, 提供给开发人员。

2) 通过启动被测应用, 检测启动crash等低级致命问题

通过对Logcat (DDMS中工具) 打印内容进行监控, 查找Java层和Native层Crash信息。

Java层Crash信息如下:

```
E/AndroidRuntime( 1857): FATAL EXCEPTION: main
E/AndroidRuntime( 1857): java.lang.RuntimeException: Unable to create
service com.sample.app.internal.protocols.ProtocolsPackService: java.lang.
RuntimeException: Unable to register protocol, service is dead
E/AndroidRuntime( 1857): at android.app.ActivityThread.handleCreateService(Act
ivityThread.java:2373)
E/AndroidRuntime( 1857): at android.app.ActivityThread.access$1600
(ActivityThread.java:130)
```

Native层Crash信息如下:

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'XXXXXXXXXX'
pid: 1658, tid: 13086 >>> com.sample.app <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 64696f7e
r0 00000000 r1 00000001 r2 ad12d1e8 r3 7373654d
r4 64696f72 r5 00000406 r6 00974130 r7 40d14008
r8 4b857b88 r9 4685adb4 10 00974130 fp 4b857ed8
ip 00000000 sp 4b857b50 lr afd11108 pc ad115ebc cpsr 20000030
```

如果Crash的Trace信息中包含被测App的包名（com.sample.app），那么这个Crash就是被测App引起的。

2.稳定性

为了测试App在各种不同机型上的稳定性，通过工具测试进行数小时测试，发现Crash问题。业界主要通过两种方法进行测试，具体如下：

1) 控件遍历测试

现在业界测试实现方法基本包含以下几个步骤。

(1) 获取当前被测App的所有控件方法见表10-2。

表10-2 获取当前被测App的所有控件方法

获取方法	适用范围	原理
UIAutomator	Android4.1 及上版本 (API 16)	通过 uiautomator dump 命令，获取当前控件 Tree 结构的 xml 进行解析获得
Hierachy Viewer	所有 Android 版本	将从 Hierarchy Server 获取的 dump 信息组成自己的控件 Tree 结构
Robotium	所有 Android 版本	详见 3.3.3Robotium 实践运用

(2) 采用某种算法遍历App中获取的控件，基于二叉树遍历算法改进而来。注意点击控件后，当前控件树发生变化。

(3) 针对遍历到的控件进行相应操作，方法见表10-3。

表10-3 对控件进行操作方法

操作方法	适用范围	原理
UIAutomator	Android4.1 及以上版本（API 16）	详见 5.3UIAutomator 实战
Robotium	所有 Android 版本	详见 3.3.3Robotium 实践运用
Input 命令	Android4.1 及以上（API 16）	运行 Android 系统自带 input 命令实现

2) Monkey随机测试

运行Android原生稳定性测试工具Monkey，通过ADB（Android Debug Bridge）实现。详见4.2 Monkey测试方法。

3.功能测试

在手机QQ浏览器（Android）项目中，搭建了一套自动化工具。通过编写功能测试自动化脚本，在内部云平台设备上运行。自动化框架如图10-3所示。

但是，在这个自动化过程中，一个难点出现了。这就是验证点如何确认的问题，如图10-4所示。

当你面对图10-5这样的测试结果，如果仅仅通过文字判断，结果是完全正确的。但是，你能承认结果是正确的吗？很显然不能。因为背景颜色发白，不符合预期。

问题的关键在于： 自动化无法验证复杂的界面颜色、布局、背景等元素。

如何破解呢？从投入产出比来看，笔者采用自动化运行，人工验证结果（截图）的半自动化方式。

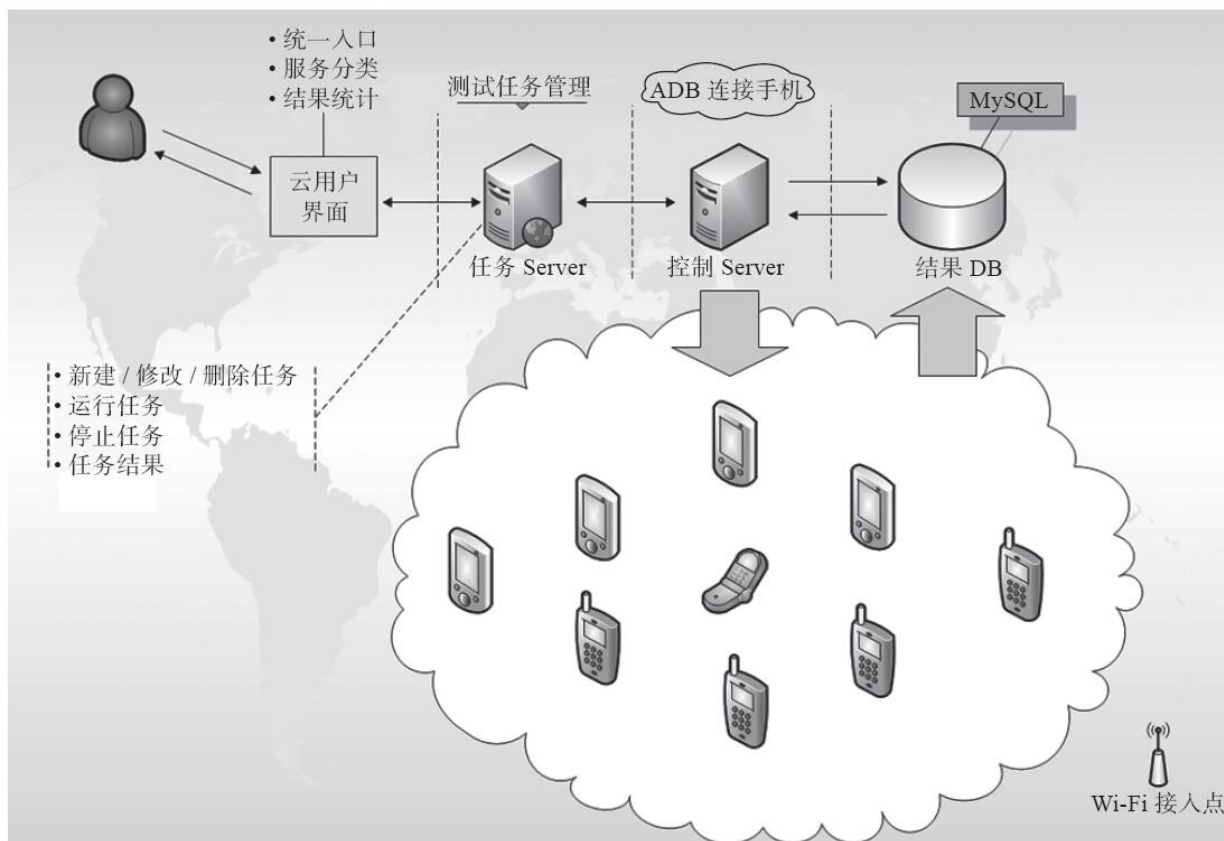


图10-3 QQ浏览器（Android）机型兼容自动化框架



图10-4 验证点的难题



最后效果如图10-5所示。



图10-5 人工半自动化验证结果

本方案有两个关键点：云平台+自动化框架。

·**云平台**：笔者选择腾讯公司内部Kapalai平台作为云平台实现。

·**自动化框架**：笔者选取腾讯公司内部最流行的Android开源框架选择QQDriver作为自动化实现技术。

QQDriver的测试用例结构如图10-6所示。

对上图所示各项说明如下：

TestSuite: 测试用例集合，类似JUnit的TestSuite方式管理用例。

Case: 测试用例，每个测试用例中可以有很多CP（CheckPoint检查点）。

CP: CheckPoint检查点，每个检查点对应一张手机截图。

Result.xml: 在这个文件中会记录Case、CP和截图之间的映射关系。这样的映射关系方便测试结果录入DB进行管理和展现。

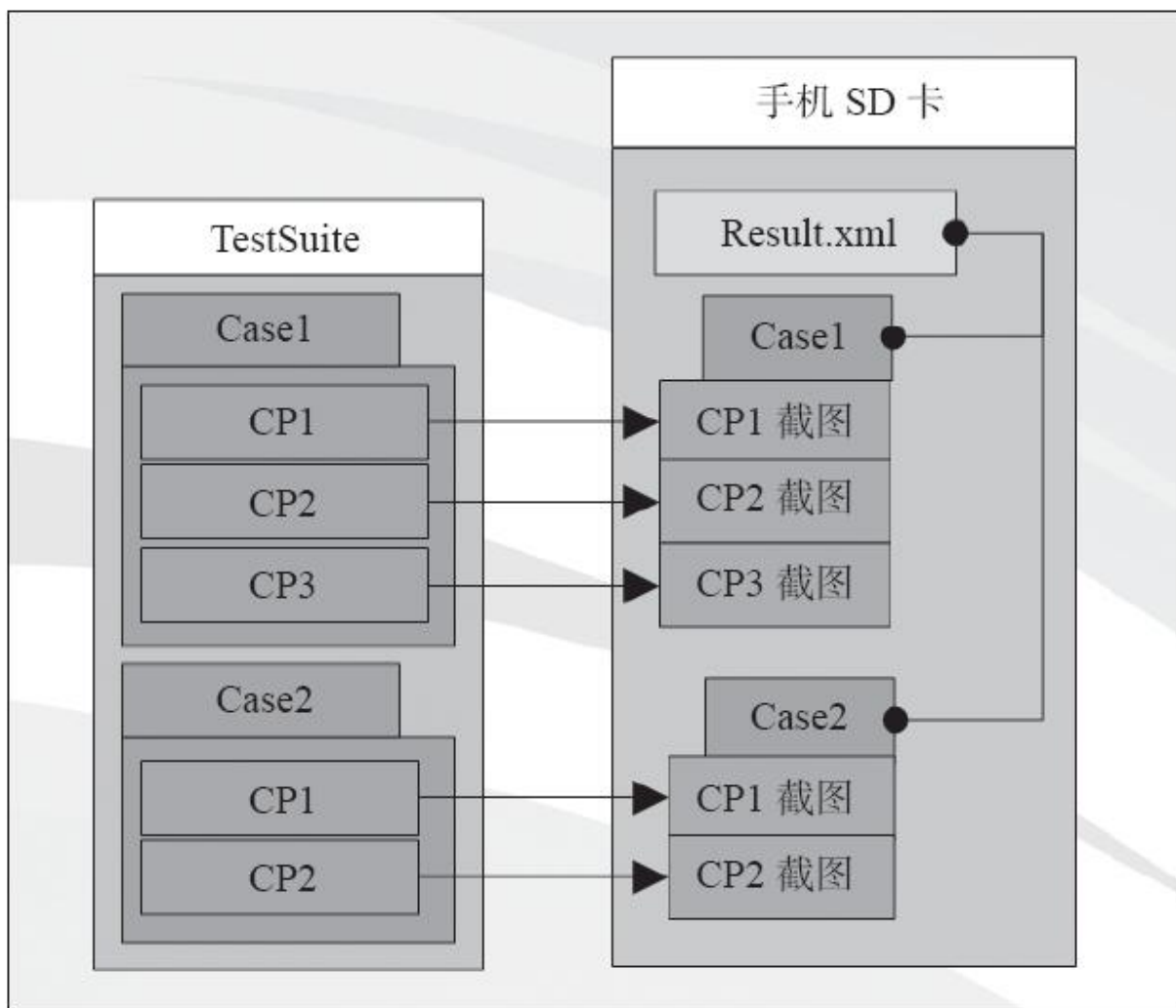


图10-6 QQDriver的测试用例结构

整个方案运行结果如图10-7所示。

在线反馈 QQ交谈

eirol退出登录

熊猫测试系统
The panda test platform

首页 机型兼容测试 覆盖安装测试 性能测试 帮助与支持

您的位置：首页 > 机型兼容测试 > 机型列表

任务ID	用户	机器	用例总数	成功用例数	失败数	未执行数	应用程序是否 crash	详情	log
		OPPO N1T	6	6	0	0	0	查看详情	下载
		Meizu M353	6	6	0	0	0	查看详情	下载
		samsung GT-I9260	6	6	0	0	0	查看详情	下载
		HTC HTC 9088	6	6	0	0	0	查看详情	下载
		samsung GT-I8262D	6	6	0	0	0	查看详情	下载
		samsung GT-I9300	6	6	0	0	0	查看详情	下载
		samsung GT-I9128	6	6	0	0	0	查看详情	下载

图10-7 整个方案运行结果

按照测试用例组织，截图半自动确认界面如图10-8所示。



图10-8 截图半自动确认界面

4) 开源自动化平台

业界主流多机型开源自动化平台，是在GitHub上的STF（<https://github.com/openstf/stf/>），如图10-9所示。

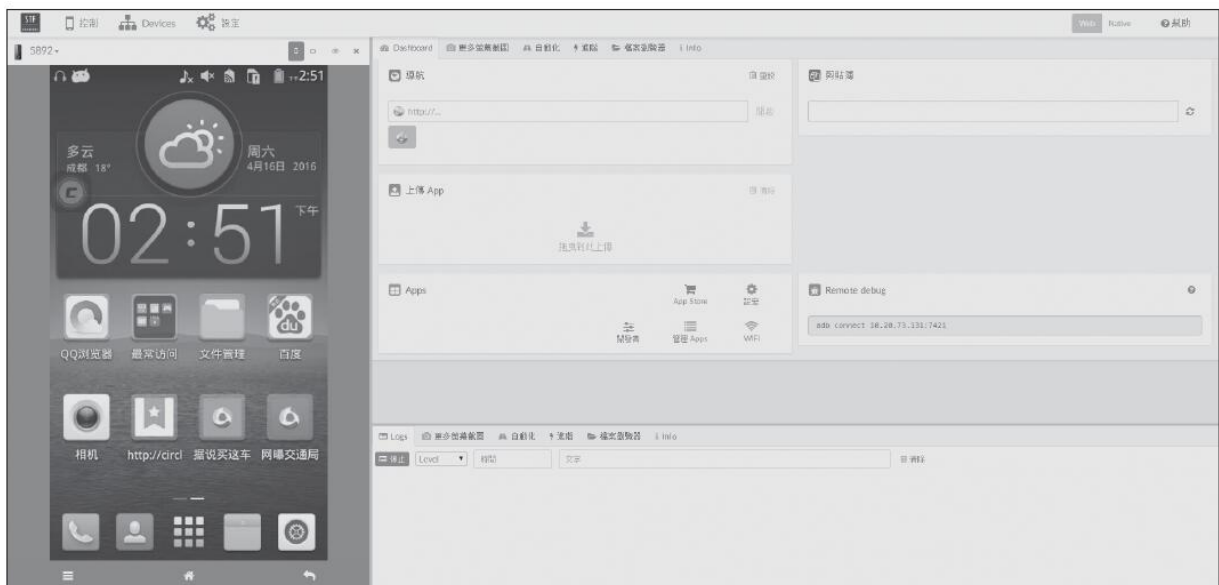


图10-9 STF主界面

其主要特性包括：

- 支持Android2.3.3（API Level 10）到Android N（API Level 23）。
- 无须手机Root。
- 支持屏幕实时操作，例如：点击、输入、拖动等。
- 拖放安装APK。
- ADB Remote Debug。
- 基于ADB Remote Debug的自动化。

搭建STF平台的步骤如下：

注意：本教程在Mac X一体机（OS 10.9）上实验通过。其他操作系统，如Linux、Windows官方暂未给出详细文档，暂不介绍。详细信息请参考官方文档安装指南。

（1）安装Brew（http://brew.sh/index_zh-cn.html）工具。在终端窗口输入命令代码如下：

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

注意：内网用户需要配置curl和git的外网proxy才能访问。

（2）安装所有依赖组件。在终端窗口输入命令代码如下：

```
brew install rethinkdb graphicsmagick zeromq protobuf yasm pkg-config
```

注意：内网用户请输入外网proxy命令代码如下：

```
http_proxy=http://<proxy-server>:<proxy-port> brew install rethinkdb  
graphicsmagick zeromq protobuf yasm pkg-config
```

（3）安装NPM（<https://www.npmjs.com/>）。在终端窗口输入命令代码如下：

```
brew install npm
```

注意：内网用户请输入外网proxy命令代码如下：

```
http_proxy=http://<proxy-server>:<proxy-port> brew install npm
```

(4) 安装STF的NPM包。在终端窗口输入命令代码如下：

```
npm install -g stf
```

注意：内网用户请输入外网proxy命令代码如下：

```
npm config set proxy http://<proxy-server>:<proxy-port>
```

(5) 点击“Download ZIP”下载最新master主线STF代码，如图10-10所示。

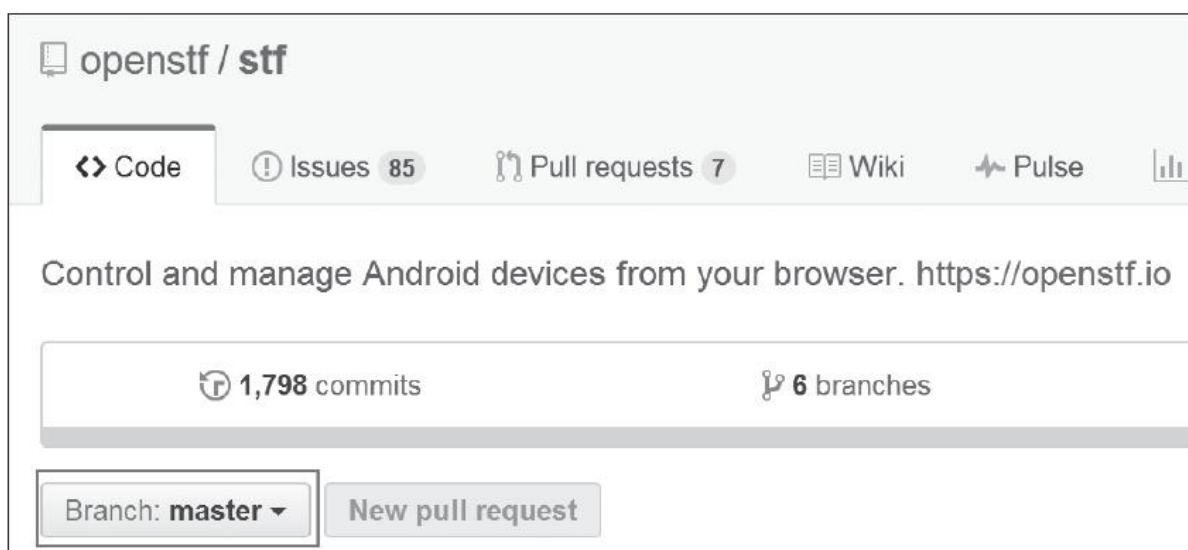


图10-10 STF代码下载

(6) 解压并进入源码根目录，进行安装。在终端窗口依次输入命令代码如下：

```
cd Downloads/stf-master/  
npm install  
npm link
```



注意 内网用户在npm link之前需要配置bower和git的proxy。

- (1) 在下载stf目录中，编辑.bowerrc。
- (2) 加入proxy: "http://<host>: <port>"。
- (3) 退出并保存.bowerrc文件。
- (4) 运行命令: `git config--global url."https://".insteadOf git: //`。
- (5) 如果这里失败，再多试几次，可能是外网连接不稳定。
- (7) 启动rethinkdb数据库。在新终端窗口输入命令代码如下:

```
rethinkdb
```

- (8) 启动STF平台。在新终端窗口输入命令代码如下:

```
stf local --public-ip <部署  
server ip>
```

- (9) 将手机通过USB连接到Mac X一体机上。注意：可以通过USB Hub扩展多个USB接口。

(10) 打开浏览器访问`http://<部署server ip>: 7100`，输入任意用户名和邮件地址登录。

(11) 在Devices界面选择一部手机，如图10-11所示。

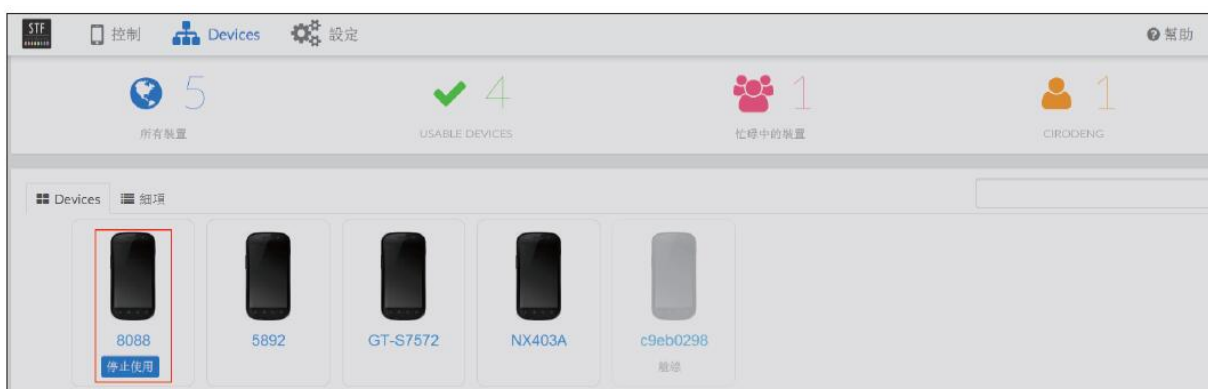


图10-11 选择一部手机

(12) 进入“控制”界面，复制Remote Debug中的值`adb connect<ip>: <port>`，如图10-12所示。

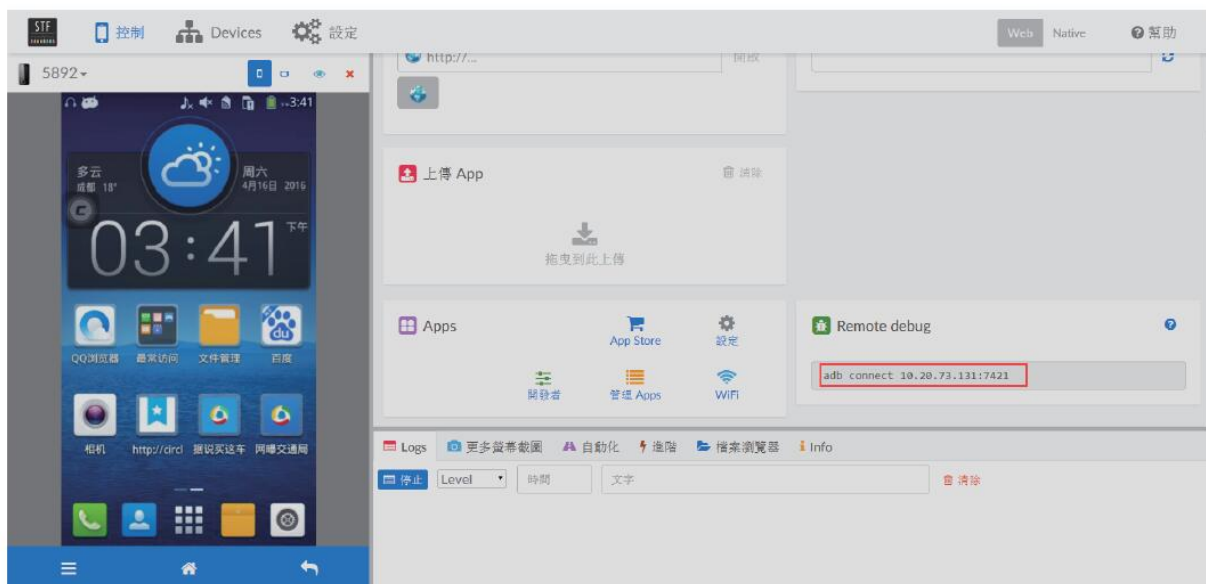


图10-12 Remote Debug值

(13) 在本地PC的CMD窗口中输入刚才复制的值“adb connect<ip>: <port>”，如图10-13所示。

(14) 在本地PC的CMD窗口中输入“adb devices”查看远程手机连接，如图10-14所示。

```
D:\>adb connect 10.20.73.131:7421
connected to 10.20.73.131:7421
```

图10-13 adb远程连接手机

```
D:\>adb devices
List of devices attached
99b7ecee          device
10.20.73.131:7421  device
10.20.73.131:7405  device
```

图10-14 adb远程连接手机

(15) 在本地PC的CMD窗口中通过adb命令可以进行远程自动化操作。例如：

·安装APK或自动化脚本：adb-s 10.20.73.131: 7405 push<安装包或者脚本包APK>。

·启动APK或自动化脚本：`adb-s 10.20.73.131: 7405 shell am start-n<包名>/.<主Activity>`。

·自动化结果可以存在手机SD卡上，通过：`adb-s 10.20.73.131: 7405 pull/sdcard/result.txt`进行拉取和验证。

（16）如果要在多种机型上操作，请重复步骤（11）~（15）即可。

10.2.3 云平台测试

本章通过介绍业界主流的云平台使用，帮助读者快速了解如何借助云平台实现兼容性测试。

1. 腾讯优测 (<http://utest.qq.com/>)

腾讯优测作为腾讯对外提供的Android手机平台，在业界有不少的用户使用。

(1) 用QQ账号登录，如图10-15所示。



QQ登录 企业帐号登录 RTX登录

帐号密码登录

推荐使用快速安全登录，防止盗号。

支持QQ号/邮箱/手机号登录

密码

登录

图10-15 腾讯优测登录

(2) 点击菜单“应用测试”进行用户信息认证，如图10-16所示。



个人资料 完善资料即可获得价值100元的测试大礼包

*真实姓名：

*手机号码：

*手机验证码：

*邮箱：

图10-16 腾讯优测用户认证

(3) 上传APK进行适配测试（请使用Chrome内核浏览器），如图10-17所示。

测试项

✓ 安装/启动/登录/执行/卸载

✓ 运行截图

✓ 安装时间/启动时间/CPU/内存

✓ 核心场景遍历

✓ log日志下载

✓ 在线报告

上传项目包进行测试

您有代金券未领取，立即领取

文件：

点我使用近期历史包，方便快捷

上传文件

微信通知：

测试完成微信通知

查看实例

图10-17 腾讯优测上传APK

(4) 点击“下一步”，进入机型选择页面，如图10-18所示。

品牌

360

华硕

步步高

酷派

朵唯

金立

Gigaset

广信

HTC

华为

联想

LG

Letv

魅族

摩托罗拉

努比亚

一加

OPPO

橙石

PPTV

华硕

ASUS_X002

_X550

_Z00UDB

步步高

图10-18 腾讯优测选择机型

(5) 点击“提交”完成任务创建，等待任务完成，如图10-19所示。

版本测试包详情					
报告编码	报告类型	安装包版本	提测时间	设备数	所有状态 
1200181	极速五十款	6.8.0.2510	2016-06-17 14:27:31	50	等待中
935705	自选机型	6.4.1.2055	2016-02-14 12:16:41	15	完成

图10-19 腾讯优测提交任务

(6) 任务完成后，查看任务结果，如图10-20所示。



缺陷分析应用测试云手机优管家优社区

西西 

账户余额: 2850元

应用测试 / 任务列表 / 任务详情



应用名称	QQ浏览器	应用大小	20M	版本号	6.4.1.2055
支持系统	Android 2.3 及以上	创建时间	2016-02-14 12:16:41	结束时间	--
测试类型	自动化测试	完成终端数	12	覆盖用户数	632万

报告概况性能分析终端详情

测试结果

全部

通过

失败

品牌

全部

 机型

全部

 系统版本

全部

品牌	机型	系统版本	测试结果	出错次数	问题描述	覆盖用户数	操作
Xiaomi	MI 4LTE	Android OS 4.4.4	通过	--	--	3万	详情 日志下载

图10-20 腾讯优测查看任务结果

(7) 点击“缺陷分析”→“适配分析”→“上传文件”，如图10-21所示。



图10-21 腾讯优测上传APK文件

(8) 上传完成后，点击“提交扫描”，如图10-22所示。

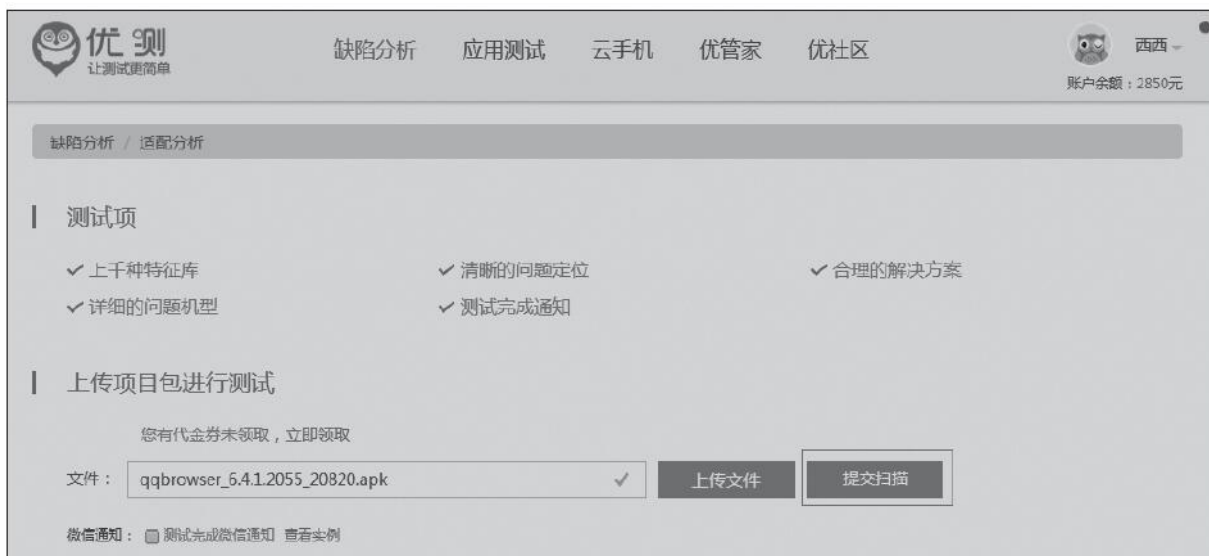


图10-22 腾讯优测提交扫描

(9) 等待任务完成。

(10) 点击“详情”查看适配分析详细信息，如图10-23所示。



图10-23 腾讯优测适配分析详细信息

2. Testin (<http://www.testin.cn/>)

(1) 注册账号登录，如图10-24所示。



The image shows a login and registration interface for Testin. It features a light gray background with a white central area. At the top, there is a text input field with a person icon and the placeholder text "请输入邮箱" (Please enter email). Below it is another text input field with a lock icon and the placeholder text "请输入密码" (Please enter password), which includes a small "SHOW" button on the right. A large "登录" (Login) button is positioned below the password field. A horizontal line with the character "或" (or) in the center separates the login section from the registration section. Below the line are two buttons: "QQ登录" (Login with QQ) and "微博登录" (Login with Weibo). At the bottom, there is a link "还没有Testin账号? 快速注册" (Don't have a Testin account? Register quickly) and a link "忘记密码" (Forgot password).

图10-24 Testin注册账号登录

(2) 点击“开始测试”，上传安装包，如图10-25所示。



图10-25 Testin上传APK包

(3) 输入相关信息，点击“下一步”，如图10-26所示。

(4) 选择“标准兼容测试”（免费），点击“提交测试”，如图10-27所示。

(5) 点击“随机100款”，其他设置如图10-28所示。



QQ浏览器 android

6.4.1.2055(Build 642055)

更换图标

应用名称

QQ浏览器

应用类别

 应用

 游戏

应用分类

系统安全 ▼

备注

下一步

图10-26 Testin输入相关信息



图10-27 Testin选择测试类型



图10-28 Testin选择机型数量

(6) 点击“返回我的测试”，点击“报告详情”，如图10-29所示。

(7) 下拉报告，点击“不兼容合计”中的数字查看不兼容机型详情，如图10-30所示。

测试记录					
应用	服务	提测时间	测试概述	测试报告	
QQ浏览器 6.4.1.2055 (Build 642055)	Android 兼容测试-随机100款(应用)	2016-02-14 12:28:58	99%	报告详情	

图10-29 Testin任务列表

行业数据：应用-系统安全 <small>NEW</small>		不兼容数据						兼容数据	
		不兼容合计	安装失败	启动失败	运行失败	功能异常	UI异常	通过	待优化
您的App水平 <small>?</small>	终端数	1	0	0	1	0	0	99	0
	影响用户数（万）	0	0	0	0	0	0	515	0
	占比（%）	1.00	0.00	0.00	1.00	0.00	0.00	99.00	0.00
行业平均水平 <small>?</small>	占比（%）	10.64	2.76	0.00	7.87	0.00	0.00	89.36	-
行业最优数据 <small>?</small>	占比（%）	0.00	0.00	0.00	0.00	0.00	0.00	100.00	-

图10-30 Testin不兼容机型详情

3.百度云（<http://mtc.baidu.com/>）

(1) 点击“服务”→“自动化测试”，如图10-31所示。



图10-31 百度云自动化测试

(2) 点击“全面兼容测试”→“Android测试”，如图10-32所示。



图10-32 百度云全面兼容测试

(3) 点击“上传App选择”上传被测App包。选择50款热门机型，点击“立即购买”（免费），如图10-33所示。

创建任务

终端类型

Android

iOS

基本信息

终端兼容测试：大量真机多维度测试，兼容性测试无死角

配置信息

* 上传App：

qqbrowser_6.4.1.2055_20820.apk 100%

选择

✓

上传用例：

待测APP测试用例

选择

注:点此下载免费工具进行测试脚本录制，如需帮助请前往>>

* 选择测试终端：

50款热门机型

100款热门机型

200款热门机型

邮件通知：

ciro.deng@qq.com

购买信息

价格：~~¥200~~ 元 （每台手机¥4元）

实际支付：¥0 元 免费试用

立即购买

图10-33 百度云上传被测App包

（4）等待任务执行完毕，点击“查看”显示全面兼容测试报告，如图10-34所示。

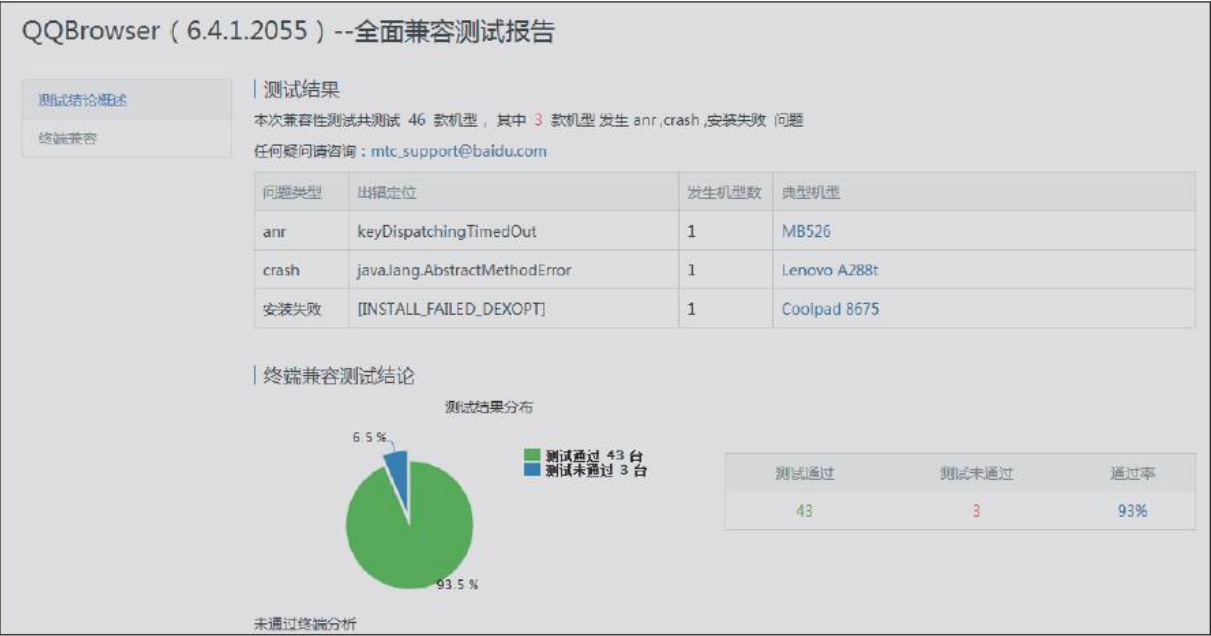


图10-34 百度云全面兼容测试报告

(5) 点击“深度遍历”测试，再点击“创建”，上传APK，如图10-35所示。

创建任务

终端类型 Android

基本信息 深度遍历测试：通过在主流真机终端上模拟真人对移动应用的UI操作行为，自动遍历控件从而发现程序的功能问题

配置信息

* 上传App：

☐ 选择最近上传的App：QQBrowser6.4.1.2055

☒ 华为P8 (android 5.0.1) 屏幕分辨率：1920*1080 CPU：八核2.0GHz 内存：3G

☒ 三星GT-N7100 (Note 2) (android 4.1.2) 屏幕分辨率：1280*720 CPU：四核1.6GHz 内存：2G

邮件通知：

购买信息

价格：¥10 元 (每台手机¥5元)

实际支付：¥0 元

图10-35 百度云深度遍历上传APK

(6) 等待任务完成，点击“详情”查看深度遍历结果，如图10-36所示。

测试结果

共执行测试项 2 个,没有发现任何问题。

基本信息

包名	com.tencent.mtt	版本号	6.4.1.2055			
----	-----------------	-----	------------	--	--	--

遍历测试

机型	系统版本	分辨率	有无黑边	有无重影	执行结果	详情
三星N7100 (Galaxy Note II)	android 4.1.2	720x1280	无	无	✔	查看
华为P8	android 5.0.1	1920x1080	无	无	⚠	查看

图10-36 百度云深度遍历测试结果

(7) 点击“查看”，查看遍历截图，如图10-37所示。



图10-37 百度云深度遍历截图

4.云平台对比

笔者对以上介绍的云平台在功能上做了横向对比，以方便读者在项目中选择合适的工具，见表10-4。

表10-4 云平台功能横向对比

特性	腾讯优测	Testin	百度云
适配分析	Y	N	N
漏洞分析	Y	Y	N
控件遍历	Y	Y	Y
手机租用	Y	Y	N
专家测试	Y	Y	Y

5.云平台收益实践

1) 典型案例

在手机QQ浏览器皮肤测试中，笔者就通过10.2.2节中功能自动化发现了问题。在应用了“幸福吃货”皮肤后，发现三星Nexus S在卡片化页面时，文字部分背景变为透明。经过业务人员测试确认，是个特殊机型Bug，如图10-38所示。



图10-38 皮肤机型问题

本方案的主要收益，源于缩短在多台手机上兼容性测试时间。通过在手机QQ浏览器（Android）上的项目实践，笔者得到如下数据。

2) 成本

自动化建设时间：2周=80小时×人

18个模块脚本编写时间：18×25=450小时×人

共计：530小时×人

由于平台建设属于固定投入，这部分收益通过长期多版本测试进行摊薄，可以忽略不计。脚本编写调试时间是主要耗时，平均每个模块需要25小时×人。当然，脚本可以通过简单修改在以后的每次版本测试中复用，预期在三次版本测试后，投资收益基本平衡。而且随着脚本编写水平的提高，每个模块的完成时间还能缩短。

3) 收益

自动化所实现的验证点，占手工测试验证点的80%，能够基本替代手工测试，如图10-39所示。

手工测试时间也明显缩短，如图10-40所示。

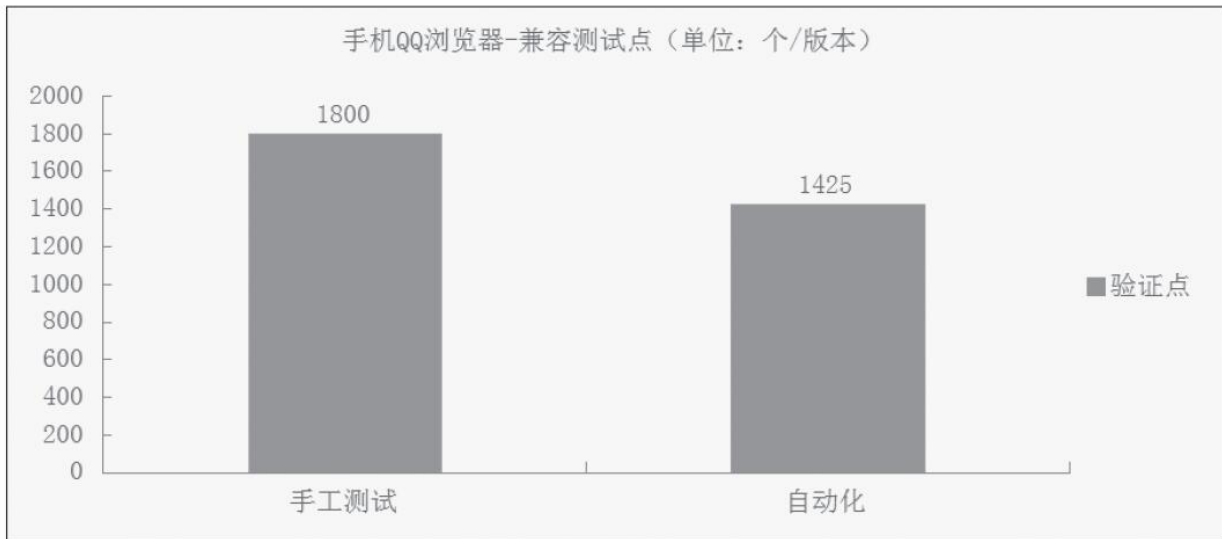


图10-39 手工测试对比自动化收益

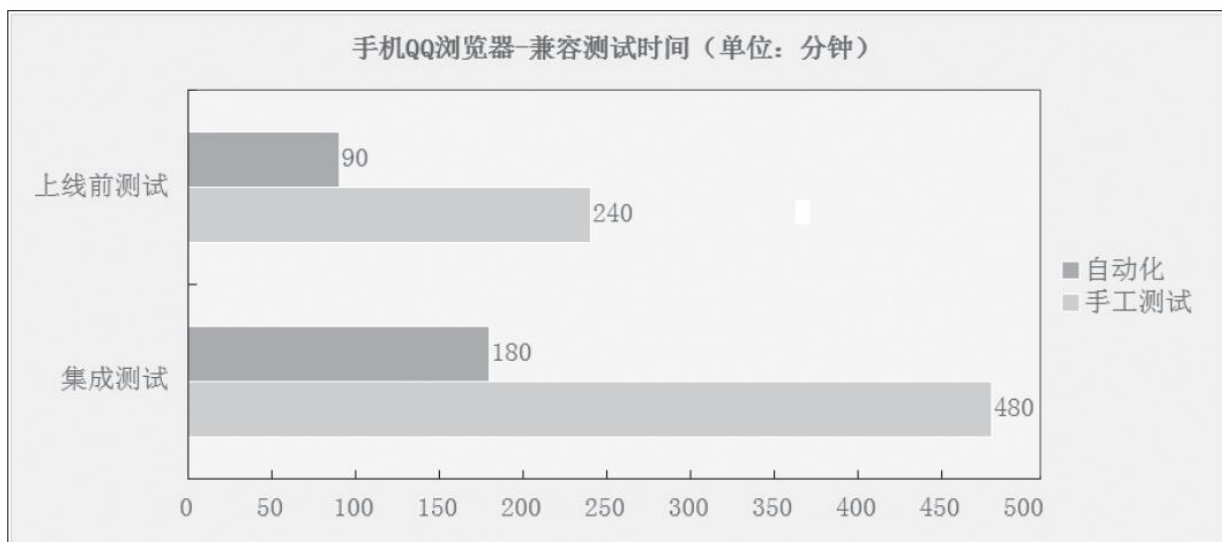


图10-40 手工测试对比自动化测试时间

手机QQ浏览器项目组在“集成测试阶段”需要测试20款手机兼容性，从上图中统计数字可以看出，通过自动化，“集成测试阶段兼容测试时间”缩短了60%，“上线前兼容测试时间”缩短了60%。

在手机QQ浏览器6.0版本测试中，通过机型兼容平台，发现Bug共计30个，占总机型Bug比例的75%，如图10-41所示。

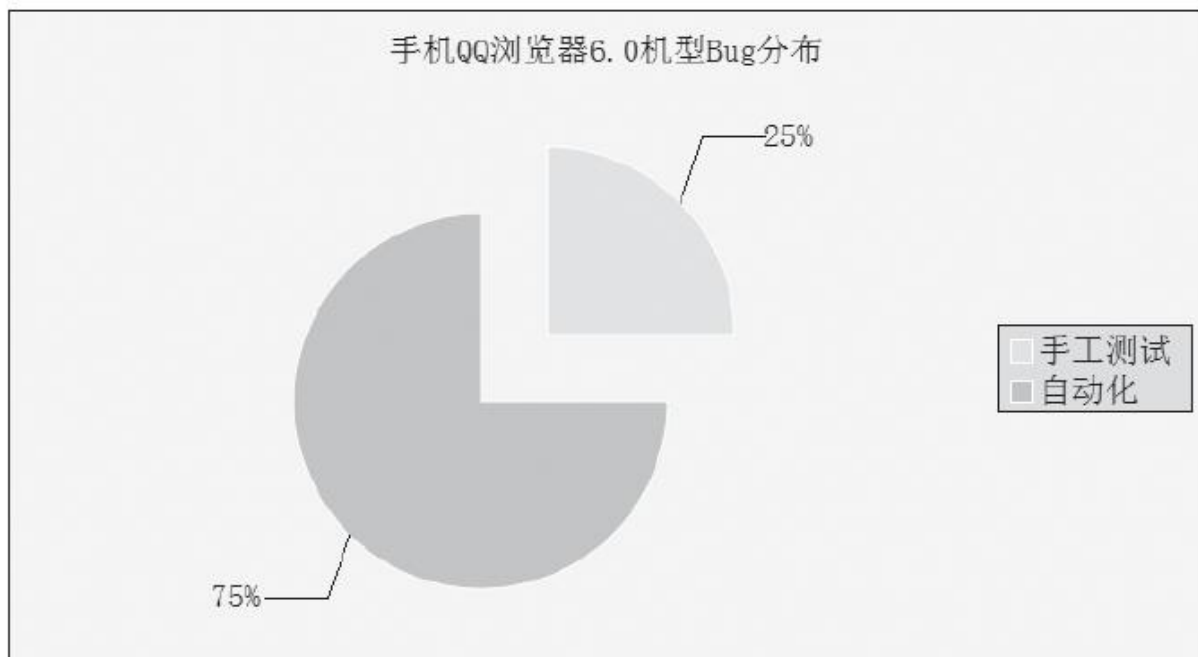


图10-41 机型兼容Bug比例

10.3 兼容性测试思考

UI级别的自动化给人的印象一直就是“变化太大，收益太低”。一旦UI发生了较大变化，之前的自动化脚本就会有较大改动，投入高，收益低。

怎么破解这个难题？思路如下：

·**降低建设成本**：笔者以编写自动化脚本为例，首先，选择一个低学习成本而且高效率的框架很重要。其次，不断地累计公共函数，让脚本开发速度提升数倍。

·**提高使用频率**：自动化测试使用频率越高，收益就越高。同一套自动化脚本，在当前版本每次回归时都能使用；同样，经过简单修改后，在下个版本中也能发挥重要作用。

·**以不变应万变**：自动化的模块还是优先选择UI相对变化较小的模块，这些是适合自动化的部分，能在未来减少变化带来的成本。

·**发展多种经营**：自动化脚本的用途，绝对不只是在功能验证上这么简单。其他各种测试都可以用到，例如：覆盖安装、性能测试、安装包验证.....发掘更多的用途就会有更大的收益。

10.4 本章小结

通过阅读本章内容，读者应该了解了兼容性测试的定义、范围以及常用的兼容性测试方法。特别是利用业界主流的云平台，读者能低成本尝试，获得不错的收益。当然，任何方法都有其弊端和不足。本章还罗列了笔者在测试过程中遇到的困难和解决方法，希望能够对读者有启迪作用。